中国科学院大学

University of Chinese Academy of Sciences

博士学位论文

面向深度学习应用的编程与编译优化技术研究

作者姓名：夏春伟

指导教师：崔慧敏 研究员

中国科学院计算技术研究所

学位类别：工学博士

学科专业：计算机系统结构

培养单位：中国科学院计算技术研究所

2022 年 12 月

Research on Programming and Compilation Optimization for Deep Learning Applications

A dissertation submitted to

University of Chinese Academy of Sciences

in partial fulfillment of the requirement

for the degree of

Doctor of Engineering

in Computer System and Architecture

By

Chunwei Xia

Supervisor: Professor Huimin Cui

Institute of Conputing Technology, Chinese Academy of Sciences

December, 2022

中国科学院大学
学位论文原创性声明

本人郑重声明：所呈交的学位论文是本人在导师的指导下独立进行研究工作所取得的成果。承诺除文中已经注明引用的内容外，本论文不包含任何其他个人或集体享有著作权的研究成果，未在以往任何学位申请中全部或部分提交。对本论文所涉及的研究工作做出贡献的其他个人或集体，均已在文中以明确方式标明或致谢。本人完全意识到本声明的法律结果由本人承担。

作者签名：

日　　期：

中国科学院大学
学位论文授权使用声明

本人完全了解并同意遵守中国科学院大学有关收集、保存和使用学位论文的规定，即中国科学院大学有权按照学术研究公开原则和保护知识产权的原则，保留并向国家指定或中国科学院指定机构送交学位论文的电子版和印刷版文件，且电子版与印刷版内容应完全相同，允许该论文被检索、查阅和借阅，公布本学位论文的全部或部分内容，可以采用扫描、影印、缩印等复制手段以及其他法律许可的方式保存、汇编本学位论文。

涉密及延迟公开的学位论文在解密或延迟期后适用本声明。

作者签名：　　导师签名：

日　　期：　　日　　期：

摘 要

各种各样的人工智能应用越来越多地使用深度神经网络。深度神经网络已经成为许多顶级应用的核心组成模块。与此同时，随着越来越多的移动端集成了智能处理器，深度神经网络模型可以被部署到云端、移动端、甚至端云协同的部署。优化深度学习应用的响应时间和能量消耗对于提升用户体验非常关键。而当前的编程框架与编译器难以有效地优化 DNN 模型在智能处理器上的性能。这是因为整个深度学习生态中，DNN 模型的架构、参数和计算量等差异巨大；智能处理器之间的架构、计算性能和功耗显著不同；深度学习的软件栈也展现出了极度复杂的特性。深度学习生态的复杂性为移动端和云端优化 DNN 模型的推理带来了巨大的挑战。

编程框架和编译器是解决深度学习应用推理性能的关键，本文以 DNN 模型在各种软硬件平台上的性能特征为基础，研究对 DNN 模型的特征刻画、编译优化和非 DNN 代码协同部署，以消除上层应用和底层硬件之间的鸿沟。本文探究了深度神经网络的软件栈，提出了一个基准测试框架刻画 DNN 模型的性能特征，并借助编译优化为 DNN 模型生成高性能的智能处理器代码，将整个深度学习应用 DNN 模型和非 AI 的代码协同部署到智能处理器上，从而优化整个深度学习应用的性能，提升智能处理器在推理场景的性能表现。

本文围绕深度学习应用在移动端和云端智能处理器上的编程和编译优化问题，针对 DNN 模型的特征刻画、编译优化和协同部署，主要的工作和贡献如下：

1. 为了刻画神经网络在不同软硬件配置下的性能特征，我们提出了一个基于基准测试的自动调优框架 DNNTune 。DNNTune 能够提供详细的 DNN 算子在不同硬件平台和软件框架上的性能和功耗。在 13 个典型的 DNN 模型上和 5 个移动平台上，一系列的基准测试发现，没有一个软件框架和硬件平台在所有的 DNN 模型上都能达到最低的延迟和功耗；并且移动平台由于功耗和体积受限，很难有效的执行 DNN 模型，特别是参数量较大追求最高准确率的 DNN 模型。为此，DNNTune 设计了两种 DNN 自动映射策略来优化性能和功耗。第一种是基于端云协同的策略，DNNTune 自动的寻找最佳切分点，并将一部分算子迁移到云端设备协同推理；第二种是基于多计算单元异构并行的策略，DNNTune 基于整数线性规划算法，自动将算子映射到不同计算单元异构并行的执行推理。在不同性能等级的智能手机上进行的测试表明，与目前最优的自动调优框架相比，DNNTune 最多达到 1.66× 的加速比，节省 15% 的能耗。

2. 为了解决 DNN 在智能处理器上的高性能代码生成的问题，我们提出了一个跨算子边界优化 DNN 推理的编译器 Souffle 。当前的 DNN 编译器不能够有效地跨算子边界发现优化机会，因而不能充分发掘潜在的性能优化机会。Souffle 基于张量表达式描述算子计算逻辑，并在计算图上全局的分析张量的依赖图和复用信息。之后，Souffle 基于数据流分析和启发式算法将整个计算图划分为子

图，并在子图上进行数学表达式等价的局部变换，从而找到一个优化的张量表达式的调度，提升指令级并行和数据复用。我们在英伟达的 A100 GPU 上衡量了 SOUFFLE 执行六个典型 DNN 模型的性能，实验结果表明，SOUFFLE 显著的超过四个当前最优的 DNN 编译器，相比厂商专门优化的 DNN 编译器 TensorRT，SOUFFLE 能够达到最高 5.17× 的加速比。

3. 为了解决 DNN 在智能处理器上的非 DNN 代码的编程难和优化难的问题，我们提出了一个面向智能处理器的数据预处理框架 GDPNPU。GDPNPU 提供了一系列简单的编程接口和库，帮助用户快速的在智能处理器上开发针对 DNN 推理数据处理程序，细粒度的处理数据中的元素。更进一步的，GDPNPU 提供了一系列的优化，可以自动的使用低精度数值完成数据预处理，并进行自动算子替换优化数据预处理的性能。在六个典型的数据预处理任务上，GDPNPU 相比 CPU 能够获得最高 6.45× 的加速比。

关键词： 深度神经网络，性能优化，编程框架与编译器，算子融合，人工智能处理器

Abstract

Deep Neural Networks (DNNs) are now increasingly adopted in a variety of Artificial Intelligence (AI) applications. DNNs have become the core building blocks of many top applications. Meanwhile, emerging AI chips are integrated into mobile devices. Thus the DNN models can be deployed in the cloud, on mobile devices, or even mobile-cloud coordinating processing. Deep learning applications' response time and energy consumption are critical for improving the user experience. However, current programming frameworks and compilers are sub-optimal in mapping DNN workloads to hardware devices and accelerators. The reason is that in the deep learning ecosystem, the architecture, number of parameters, and computation complexity of DNN models show significant differences. The architecture, peak performance, and energy consumption exhibited prominent diversity between the hardware accelerators. The overwhelming complexity of the underlying mobile ML system stack makes it a big challenge to optimize the DNN inference on mobile and cloud devices.

The programming framework and compiler is the key to optimizing the deep learning application performance. In this paper, based on the performance characteristics of DNN models on various software and hardware accelerators, we investigated the benchmarking of DNN models, compilation optimization techniques, and coordinating DNN deployment with non-AI programs, to bridge the gap between the deep learning applications and the underlying hardware. We studied the software stack of deep neural networks, proposed a benchmarking framework to show the performance characteristics of DNN models, presented a compilation optimization technique to generate high-performance code for hardware accelerators, and coordinately deployed the DNN models with non-AI programs on hardware accelerators, optimized the end-to-end deep learning applications and increased the inference performance for hardware accelerators.

This paper explored the programming and compiling optimization technologies for DNN inference by benchmarking the DNN models, compilation optimizing and coordinating deployment, including three contributions as follows:

1. To characterize the performance of DNN models under diverse software and hardware configurations, we proposed a DNN tuning framework, i.e., DNNTune, which can provide layer-wise behavior analysis across a number of platforms. Using DNNTune, this paper further selected 13 representative DNN models, 5 mobile devices ranging from low-end to high-end to characterize the DNN models. Experimental results showed that there is no such framework and hardware accelerator that can achieve the best latency and energy consumption on all DNN models. And limited by power consump-

tion and volume size, mobile devices are insufficient in executing large models for the highest accuracy. Inspired by the benchmarking results, DNNTUNE integrated two automatically mapping strategies to optimize the latency and energy consumption. The first strategy is based on mobile-cloud computing, DNNTUNE automatically finds the optimal partition point and offloads the second part of the DNN model to the cloud for execution. The second strategy is based on the parallel execution of multiple heterogeneous computing devices. Based on the integer linear programming algorithm, DNNTUNE can schedule operators to different computing devices and executes operators in parallel. We evaluated the performance of DNNTUNE on different classes of smartphones, DNNTUNE can achieve up to 1.66× speedup and 15% energy saving compared with state-of-the-art works.

2. To solve the problem of generating high-performance code for DNN operators, We presented SOUFFLE, an open-source compiler to optimize DNN inference across operator boundaries. SOUFFLE creates a global tensor dependency graph using tensor expressions, from which it traces data flow and tensor information. It leverages the data-flow analysis and heuristics to partition tensor expressions into subprograms. Within a subprogram, SOUFFLE performs local optimization via semantic-preserving transformations, finds an optimized program schedule, and improves instruction-level parallelism and data reuse. We evaluated SOUFFLE using six representative DNN models on an NVIDIA A100 GPU. Experimental results showed that SOUFFLE significantly outperforms four state-of-the-art DNN optimizers by delivering a speedup of up to 5.17× over TensorRT –the NVIDIA-tuned inference library.

3. To solve the problem of effectively pre-processing input and output data of DNN models, we presented a data pre-processing framework GDPNPUto help programmers to write and optimize data transformation programs. GDPNPUaccepts data-flow programs and can perform the computation with a low-precision numerical value. Furthermore, GDPNPUperforms operator substitution to automatically optimized the data pre-processing programs. Our experimental results demonstrated that GDPNPUcan achieve up to 6.45× speedup and even can perform complicated image processing tasks on AI accelerators.

Key Words: Deep Neural Networks, Performance Optimization, Programming Framework and Compiler, Operator Fusion, AI Accelerators

目 录

图目录

表目录

第 1 章　绪论

1.1　研究背景

最近几年，深度神经网络已经在许多领域被广泛的应用，包括语音识别、图像分类、神经机器翻译、自动驾驶等。多种多样的 DNN 网络被设计出来，包括卷积神经网络[1–5]，循环神经网络[6,7]、多层感知机[8] 和 Transformer[9] 类模型等。深度学习模型在多种任务上都取得了举世瞩目的成就。CNN 模型在 ImageNet 图像分类任务上可以取得超过 80% 准确率。以 BERT[10] 为代表的深度学习模型可以进行多种语言的翻译，甚至可以回答开放式的问题[11]。以 NeRF[8] 为代表的神经网络渲染模型可以根据多张图片就能进行物体在三维空间中的渲染。AlphaFold[12] 可以预测蛋白质的结构。基于循环神经网络的推荐系统模型[13,14] 更是在 Facebook、Google 和阿里巴巴等公司的数据中心中大量的使用，进行分类、推荐等机器学习任务。DNN 模型愈来愈多的集成在我们生活中的应用内，并且已经成为顶级应用（例如淘宝、抖音等应用）中核心的功能模块[15,16]。我们已经看到深度学习的应用在许多领域产生了颠覆式的创新。目前，深度学习仍然在如火如荼地发展。

DNN 模型的执行通常可以被分为两个阶段：训练和推理。训练阶段通常包括两个步骤，首先通过 DNN 模型前向执行计算与真实值的误差，之后基于反向传播[17] 更新 DNN 模型的权值。当训练完成后，模型的权值已经固定下来，之后部署到实际的生产环境中，进行**推理**过程。根据最近北大和微软亚洲研究院的一项研究统计[15,16]，占据了 Google 应用商店 11.9% 下载量的顶级应用都在使用深度学习模型。在 3 个月的调研时间中（2018 年 6 月到 9 月），使用深度学习的应用的数量增加了 27%，并且其中 81% 的应用以深度学习作为其核心功能。而根据 bankmycell 的统计[18]，全球有 66.48 亿的智能手机用户。因此，支持 DNN 模型进行高效的推理任务具有非常重要的意义。而根据 Facebook 的研究表明[19]，手机芯片具有极大的多样性，没有标准的手机片上系统（System on Chip，SoC）可以进行针对性的优化。前 50 个最常见的手机 SoC 也只占据了市场份额的 65%。面向如此多样的 SoC 进行 DNN 模型的编程和优化是一项巨大的挑战。在云端，Facebook 的一项研究[20] 也表明 DNN 模型在 Facebook 数据中心中已经广泛的部署。其他的数据中心，包括谷歌云、亚马逊云、商汤科技等也大规模的部署 DNN 模型，以提供多种人工智能应用的服务[21]。这一现状表明，无论是在像智能手机设备这样的移动端，还是在以数据中心为代表的云端，DNN 模型都得到了非常广泛的部署。根据 AppFlyer[22] 的统计，一秒钟额外的延迟可能会导致 7% 的用户流失，40% 的用户可能会切换到竞争对手的应用程序。因而，有效的优化 DNN 模型的推理是一个非常重要的问题。

鉴于 DNN 模型的广泛部署以及对其高效推理的巨大需求，众多的商业公司和学术团队从硬件、软件、算法多个方面对 DNN 模型进行优化，提出了一系列

面向 DNN 优化的硬件加速器、加速库、编译器、编程框架和算法。由于 DNN 模型对计算量的巨大需求[23]，研究人员开发出了多种专用硬件加速器（例如寒武纪的 MLU[24]，谷歌的 TPU[25]，英伟达的 GPU[26,27] 等，以下统称为智能处理器）以专门加速 DNN 运算。与传统的 CPU 相比，这些智能处理器具有极高的算力。例如，英伟达 A100 GPU 具有 624 TFLOPS 的 FP16[28] 峰值性能。然而，智能处理器的微体系结构和 CPU 差异巨大，甚至智能处理器之间的架构都各不相同。例如，TPU 采用脉动阵列的方式加速矩阵运算，而英伟达 GPU 则是以张量核（Tensor Core）加速矩阵运算。架构差异的智能处理器为用户的编程、性能优化以及部署带来了严重的挑战。

在应用层面，DNN 模型的架构也愈来愈复杂，计算量也越来越大。2012 年的 AlexNet[29] 只有 8 层，而 2019 年提出的 EfficientNet-B7 已经有 813 层。并且根据 Hestness[23] 等的估计，未来深度学习的模型的准确率要想达到超过人类的水平，数据集需要增长 $33-971\times$, DNN 模型将需要增长 $6.6-456\times$。总结而言，上层复杂多样的 DNN 模型应用与底层架构各异的智能处理器为开发人员的编程、优化与部署带来了严重的挑战。

1.2 研究现状

本节主要描述优化 DNN 推理相关工作的概况，并说明我们的研究工作在整个研究领域中所处的位置。和本文密切相关的研究工作的具体梳理在第 2章。本文将主流的研究工作从上到下分为如下的如下的层次：算法、编程框架、编译器与库、硬件加速器。当然，各层次之间并不是相互独立的，许多工作都需要各个层次之间的协同设计与实现。通常，经过算法优化过的 DNN 模型需要编程框架加载运行，编程框架依赖于编译器和库生成或者调用算子的二进制代码，最终运行在智能处理器上。在此做一个简单的分类。

算法。在算法层级的设计主要是在保证一定准确率的前提下，通过减少模型的参数和计算量来达到优化推理性能的目的。这方面的工作主要包括：

- 参数剪枝，将参数中的某些数值置零，用稀疏化的方式表示和运行 DNN 模型，减少运算的次数[30,31]；
- 模型量化，使用低精度数值（例如 8 位整数 INT8 代替浮点数）来表示权值，硬件计算单元一次可以计算更多的元素[32]；
- 轻量化结构设计，用计算量更低的算子代替计算量高的算子（例如可分离卷积 Depthwise convolution 代替标准的卷积算子）[33–35]；
- 模型蒸馏，用多个大模型“教”一个小的模型，从而实现模型压缩[36]。
- 自动模型架构搜索，通过机器学习的方法自动搜索模型架构，可以训练出不同计算量不同准确率的模型[4,37]。

编程框架。编程框架是 DNN 模型的核心基础设置。程序员通过编程框架提供的编程接口来搭建并运行 DNN 模型。在编程框架中 DNN 模型抽象为一个有向无环

的数据流图，其中节点是算子，表示张量具体的计算；节点之间的有向边表示张量的流动方向和算子之间的依赖关系。典型的深度学习框架包括 TensorFlow[38] 和 PyTorch[39]。此外。也有越来越多的面向移动设备的推理框架被开发出来，例如 TFLite[40]、MACE[41] 和 MNN[42] 等，为 DNN 在移动设备上的执行提供了支持。面向移动设备的编程框架也可以使用异构计算单元执行 AI 应用，例如 TFLite[43] 可以通过 GPU 代理在移动平台的 GPU 上进行 DNN 模型的推理，或者通过安卓神经网络接口（The Android Neural Networks API，NNAPI）。MNN[42] 可以通过 OpenCL、OpenGL、Vulkan 或者 Apple Metal API 来调用 GPU 执行 DNN 模型。这些移动框架支持在不同的计算平台上运行 DNN 模型。编程框架优化 DNN 模型推理的工作包括：

- 数据的加载与处理；
- DNN 模型的性能基准测试[44]；
- 计算图的优化，例如 TensorFlow 的 Grappler 模块可以执行[45] 常量折叠、算术化简等优化；
- 算子替换[46]，将算子进行等价替换，选择性能更优的算子从而达到更优性能；
- DNN 模型的算子映射，将 DNN 模型映射到不同的计算单元或者设备，例如端云协同的执行[47] 或者异构并行[48,49]；
- 自动参数调优。通过自动调节多种软件参数选择最优部署策略。

编译器与库。编程框架主要为程序员提供简单易用的编程接口，以快速的开发和部署模型。但是编程框架依赖于框架开发人员编写算子库，随着 DNN 模型和智能处理器都越来越复杂，依靠手工开发算子会消耗大量的人力和时间。深度学习编译器可以快速的为算子生成高性能的代码，是编程框架和智能处理器之间的桥梁。典型的 AI 编译器包括 TensorFlow XLA[50]、TVM[51]、nGraph[52] 等。AI 编译器对 DNN 推理的优化可以大致划分为前端优化和后端优化[53]：

- 前端优化。与编程框架中的优化类似，主要集中在计算图或者较高层级的中间表示做变换，包括算术化简、公共子表达式消除、静态内存分配和数据布局变换等；
- 后端优化。通常会考虑硬件架构的一些特性，例如访存延迟的掩盖、算子融合、面向循环变换的自动调优等。

此外，有许多硬件厂商也发布了针对自家智能处理器专门优化的算子库，例如英伟达（NVIDIA）的 cuDNN[54] 和安谋（ARM）的 ARM Compute Library[55] 等。

智能处理器。除 CPU 外，专门面向 DNN 优化的加速器，包括 GPU、FPGA 以及领域定制的芯片（Application Specific Integrated Circuit，ASIC）。其中，GPU 可以分为云端（例如 NVIDIA GPU）和移动端（例如 ARM 的 mali 系列 GPU）。典型的领域定制芯片包括寒武纪的 MLU[24,56,57]，谷歌的 TPU[25] 和华为的 NPU[58,59] 等。智能处理器通常提供了编程语言来供上层的编译器和编程框架生成优化的算子实现。例如 NVIDIA 的 GPU 提供了 CUDA[60] 编程语言，谷歌 TPU 可以通

过 XLA 编译器[50] 来调用。

面向 DNN 推理的优化在各个层次并不是孤立的，有许多工作都进行了跨层次的协同优化。例如 PatDNN[31] 结合算法层的稀疏性和编译器协同设计；PET[61] 利用 DNN 算子的近似计算再恢复进行编译器代码生成与优化；Rammer[62] 则考虑了 DNN 算子间并行性与编译器的后端代码生成。

本文工作的定位：作为计算机体系结构方向的研究人员，本文的工作主要集中在编程框架与编译器的层面，对深度学习应用进行 DNN 的性能刻画、在编译器中进行编译优化以及其中非 AI 部分的程序优化。

1.3 DNN 推理优化面临的挑战

虽然研究人员已经对如何优化 DNN 模型推理的性能在算法、软件和硬件都展开了一定程度的研究，但是目前在智能处理器编程框架面对多样的设备依然存在优化难的问题。针对多样的 DNN 模型编译器的性能优化距离硬件的峰值性能依然有很大的差距，DNN 推理的编程框架和编译器的性能优化仍然面临着以下关键问题和挑战：

如何刻画 DNN 的性能特征。模型推理的延迟与 DNN 模型本身的架构、参数量、软件框架和硬件性能密切相关。更糟糕的是，移动设备和云端设备在硬件的性能、设备制造商提供的库和处理的框架方面差别巨大。其中，移动设备的功耗也和 DNN 模型、硬件性能、网络环境紧密相关。面对如此多的影响性能和功耗的因素，如何根据用户目标（例如功耗最低或延迟最低）来选择最优的运行参数是一个关键问题。更近一步的，如何优化 DNN 模型在编程框架上的性能也是一个严重的挑战。根据 Facebook 发布的报告[19]，目前几乎所有的 DNN 的推理都是运行在 CPU 上并且其中大多数都是低端设备，只有 11% 的安卓设备其 GPU 性能是 CPU 性能的三倍以上。在移动端，将深度学习应用中的一部分任务迁移到云端是有效地降低延迟的方式[16]。如何将 DNN 模型划分并映射到移动端和云端协同执行，选取最优的配置以降低延迟和功耗是重要的问题。进一步的，移动端设备集成了多个异构计算单元。在 MLPerf 的文章中强调，探索加速器的并行性是一个重要的问题[44]。而软件框架和智能处理器的差异又使得将 DNN 算子有效地映射到多个异构设备非常困难。如何充分利用移动端设备上计算资源加速 DNN 的推理，协同调度移动设备的异构计算单元，并行执行 DNN 的推理是一个很大的挑战。

如何为 DNN 生成性能优异的代码。编程框架依赖于编译器或者算子库进行推理。单纯的优化编程框架已经难以进一步提升 DNN 推理性能。DNN 模型中多样的算子与架构各异的智能处理器之间性能优化的鸿沟愈来愈大。深度学习编译器旨在解决在智能处理器上快速生成高性能代码的问题。现有的工作在面向单算子代码生成已经做了许多工作[63–65]。由于算子的数量非常多（例如 TensorFlow 中有将近 800 个核心算子），不同算子之间的构成的小计算图会产生组合爆炸。同时对多个算子进行高效的代码生成依然是一个很难但重要的问题。

智能处理器通常具有极高的算力（数十乃至数百 TFLOPS）和很高的性能功耗比(可达到数百 GFLOPS 每瓦特)，因而访存会成为影响 DNN 推理的瓶颈。如何在多算子间进行数据复用是提升 DNN 推理性能的关键。基于模式匹配和基于规则的方法面对多样化的算子组合很容易错失算子融合的机会，而基于编译分析的方法又不能从全计算图的角度发掘优化机会。因此，如何从多层级分析 DNN 模型同时考虑多个算子进行代码地生成与优化是一个重要的问题。

如何高效地进行数据预处理。智能处理器和上层软件的协同工作极大地加速了 DNN 模型的推理速度，但是 DNN 模型只是深度学习应用中的一部分，深度学习应用中还存在着非人工智能的程序(例如图像预处理、特征匹配等)，与 DNN 模型共同构成了深度学习应用的流水线。在一些场景中，CPU 进行数据预处理和后处理会成为新的性能瓶颈[66–68]，导致 CPU 忙碌而智能处理器空闲的情况发生。目前的数据预处理主要依赖于 CPU 上的库，例如 OpenCV[69]、NumPy[70]等。这些库经过了高度的优化，难以继续提升性能。如果将部分数据处理的任务迁移到加速器上执行，则有希望减少数据搬运的开销，降低 CPU 的负载并提升智能处理器的利用率。然而，目前智能处理器通常只能加速 DNN 模型和做一些编解码的工作，没有提供编程框架以进行数据处理。此外，DNN 模型通常为计算密集型应用，而数据预处理通常是访存密集型且需要细粒度的操作数据元素。如何针对深度学习应用中非 AI 部分的程序（例如数据预处理程序）的特点，提供编程接口并有效地优化其在智能处理器性能是一个值得探索的问题。

1.4　本文的主要内容和创新点

1.4.1　研究内容

围绕当前深度学习应用推理优化所面临的关键挑战，本文分别从编程框架与编译器两个层次展开研究，从而尝试解决深度学习应用推理中的 DNN 性能特征刻画、DNN 代码优化和数据处理优化的问题。本文首先在编程框架的层次提出一个自动进行 DNN 模型基准测试和调优的框架，从而支持在移动设备和云端平台自动地在多种软件框架和配置下衡量和分析 DNN 模型的性能和功耗；基于基准测试得到的一些核心观察，本文提出了端云协同运行 DNN 模型推理，以及在多个异构计算单元上并行运行 DNN 模型两种调优方法，以加速 DNN 模型在编程框架上的性能；更进一步地，本文在深度学习编译器的层次提出了一个基于张量表达式（Tensor Expression，TE）的全局分析和局部变换的深度学习编译器，为 DNN 模型推理生成高性能的代码；最后，针对 DNN 模型部署过程中对非 DNN 部分的代码优化，本文提出了一个编程框架，以在智能处理器上进行输入输出数据高效地处理。本文研究的具体内容包括：

(1) 由于移动端平台存在着严重的碎片化问题，并且不同的软件框架在不同参数配置下性能和功耗各异，DNN 模型在部署时如何选择最优的参数配置给用户带来了很大的困难。针对 DNN 模型性能特征刻画的问题，研究如何自动的对 DNN 模型进行基准测试，以及如何选择最优的软硬件配置进行性能优化。更

进一步地，现有的编程框架不能自动将 DNN 模型映射到多个计算设备（包括云端智能处理器和移动端智能处理器）执行，从而难以进一步提升推理性能。为此，本文提出了一个自动基准测试与调优框架，框架可以根据用户选择的目标，在不同的软硬件平台上以多种软硬件配置上刻画 DNN 推理的特征；程序员只需要设置少量的参数，就可以将 DNN 模型自动映射到云端平台进行端云协同的执行，或者映射到多个计算单元进行异构并行执行，优化推理性能。

(2) 针对 DNN 模型在智能处理器上代码生成与优化的问题，研究如何进行高效的 DNN 代码生成，提高 DNN 模型的推理性能。当前基于算子融合的工作不能从整个 DNN 模型架构的视图全局分析数据复用机会，不能够跨计算密集型算子进行算子融合与指令调度，从而难以发掘全局优化的机会。为此，本文提出了一种基于全局分析和局部变换的 DNN 代码生成方法，从整个 DNN 模型的视图分析张量的数据复用、活跃区间等信息，并在局部的算子之间进行基于张量表达式的等价变换，有效的提升了生成代码的数据复用，降低了计算核启动的开销，从而实现了对智能处理器片上缓存的更好利用，提升了 DNN 模型端到端的推理性能。

(3) 针对深度学习应用中非 DNN 代码的在 CPU 上处理可能成为性能瓶颈的问题，研究如何在智能处理器上对非 DNN 代码进行编程与性能优化。在进行 DNN 推理时，输入给 DNN 的数据通常要进行预处理，且输出的数据也需要经过其他非 AI 程序的流水线，从而实现应用端到端执行。通常，数据的预处理在 CPU 上进行，然后传输到智能处理器上进行 DNN 推理。CPU 对数据的处理速度可能会成为整个深度学习应用的瓶颈，并且会引入额外的数据传输开销。为此，本文提出了一个面向智能处理器的数据预处理编程框架。该编程框架提供了简单的编程接口，程序员只需要编写 TensorFlow 算子代码，就能将数据处理代码运行在智能处理器上。进一步的，编程框架能够将程序员的数据预处理算法自动地进行计算图级别的优化，并自动选择合适的精度，降低数据传输与访存开销，从而提升深度学习应用中数据处理的效率，降低 CPU 的负载。

1.4.2 创新点

与上述的研究内容相对应，本文的主要贡献和创新如下：

(1) 设计了面向移动端和云端的自动基准测试与调优框架 DNNTune 。该框架由两部分组成，上层为基准测试模块，底层为自动调优模块。其中，基准测试模块根据用户的配置文件，移动平台和云端平台分别调用指定的框架来执行 DNN 模型，收集每层的执行信息。自动调优模块提供了端云协同的优化和异构并行的优化两种调优模式。将 DNN 模型自动地映射到云端或者异构智能处理器上进行加速，最终根据用户的目标选择最优部署策略。实验结果表明，基于 DNNTune 的 DNN 推理性能相比现有的系统能够达到最高 1.66× 的加速比，最多节省 15% 的能耗。

(2) 设计了基于全局分析和局部变换的跨算子边界优化的编译器 Souffle 。Souffle 能够在整个 DNN 的计算图级别进行全局的分析，寻找张量在时间维度

和空间维度的复用，从而可以发现更多数据复用机会；同时在算子层级基于张量表达式进行元素级别的依赖关系分析，并局部地进行张量表达式数学等价的变换。Souffle 还利用全局同步原语，使得计算密集型算子也能够融合到一个计算核（Kernel），执行跨算子边界的指令级联合调度优化，为 DNN 模型生成了性能更优的代码，弥补了目前算子融合方法的缺陷。实验表明，Souffle 显著优于目前的解决方案，与厂商优化的推理库 TensorRT 相比，可达到最高 5.17× 加速比。

(3) 提出了面向 DNN 推理数据预处理的编程框架 GDPNPU。GDPNPU 利用已有深度学习框架的编程接口，将原本需要在 CPU 上处理的数据的迁移到了智能处理器上进行处理，缓解 CPU 处理成为瓶颈的问题，拓展了智能处理器的用途。具体地，GDPNPU 允许用户基于 TensorFlow 来写数据预处理的程序，通过调用 GDPNPU 提供的接口，即可将数据预处理程序运行在智能处理器上，从而降低了 CPU 的工作负载和主机与智能处理器之间数据传输的开销。GDPNPU 可以自动对数据预处理程序进行智能处理器性能感知的计算图优化，通过算子替换来调用在智能处理器上性能更优的代码；更进一步的，通过自动精度来进一步地加速算子的计算性能。与当前的工作对比，GDPNPU 能够获得最高 6.45× 加速比。实验结果也表明，原本在 CPU 和 GPU 上才能运行的图像处理同样可以在智能处理器上运行，进一步的拓展了智能处理器的用途。

1.5 论文组织结构

本文的主要内容包含六个章节，其他章节组织如下：第 2章介绍相关工作，分别与三个研究点相对应。在编程框架层，重点介绍了当前面向 DNN 模型的基准测试、端云协同的编程以及异构并行的算子映射等相关工作。在深度学习编译器的优化方面，主要介绍了算子融合相关工作的发展脉络和面向单个算子的编译优化。在数据预处理方面，主要介绍了当前面向深度学习应用的图像处理框架，以及将非 AI 的应用运行在智能处理器上的前沿研究探索。

第 3章介绍了面向 DNN 的性能分析与自动调优的框架 DNNTune 。本章首先介绍了 DNNTune 为用户提供的配置接口，之后介绍了在多种不同类型的 DNN 模型上基准测试的结果。根据基准测试结果的核心观察，介绍了两种提高 DNN 推理性能的映射方法：基于端云协同的 DNN 推理和基于多计算单元异构并行的推理两种方法。并分别介绍了这两种方法在典型的移动平台上的性能，对实验结果进行了深入的分析。

第 4章介绍了面向智能处理器的自动 DNN 代码生成与优化的编译器 Souffle 。本章首先以 BERT 模型为例介绍了研究动机与核心观察，之后介绍了如何全局地在 DNN 模型上做分析，并局部地进行张量表达式的变换。最后衡量了 Souffle 在六个典型的 DNN 模型上的性能。

第 5章介绍了面向 DNN 数据预处理的编程框架 GDPNPU。本章首先介绍了在智能处理器上进行数据预处理的重要性，之后介绍了编程框架简单易用的编程接口。针对智能处理器的微架构特点，本文设计了面向智能处理器的计算图优

化，以及自动精度优化。最后衡量了一个典型的智能处理器昇腾 310 在多种数据预处理任务上的性能表现。

第 6章进行了全文工作的总结，并对未来 DNN 推理的编程与编译优化的工作方向进行了展望。

第 2 章　相关工作

2.1　DNN 基准测试与性能调优

2.1.1　DNN 基准测试

要优化 DNN 模型的性能，首先需要刻画各种 DNN 模型在端云平台上的性能特征，以此来指导后续的部署和优化。国内外研究人员针对 DNN 的基准测试做出了许多探索。

主要面向移动端的基准测试（benchmark）有以下的相关工作。北京大学的研究人员[71] 调研了 2019 年安卓应用使用 DNN 的情况，研究显示，安卓的应用越来越多的采用 DNN 模型作为其核心的功能。他们研究了谷歌商店中下载量最大的 16500 个移动应用，希望发现使用了 DNN 模型的应用特点，和它们用 DNN 模型的原因以及使用的 DNN 类型。研究结果发现下载最多的顶级应用都在采用 DNN，并且将 DNN 作为其核心功能的组成模块，其中美颜类的应用是使用 DNN 最多的。Hadidi 等[72] 基于 Tensorflow 和 PyTorch 基准测试了 16 种常见 CNN 网络在 Raspberry 3b、Jetson TX2、Jetson Nano 、Edge TPU 几种商业移动设备上以及服务端志强系列 CPU 等的性能表现。此外，Hadidi 等还比较了不同框架的性能，并测量了移动设备运行 DNN 模型的能量消耗和表面温度。分析发现，不同的移动端设备，其性能差别非常大，最大可达 30×；不同的框架执行 DNN 推理的性能差别很大。但是这个工作只支持 CNN 的网络，而不支持 RNN、MLP 等类型的网络，从而限制了其负载的普遍性和代表性；此外，其不支持智能手机设备。Ignatov 等[73] 提出了一个测试手机 AI 基准性能的应用：AI Benchmark。作者主要针对目前安卓平台的硬件和推理框架做了调研总结，并且测试和分析了 7 个针对图像处理 CNN 的网络在目前主流的安卓手机平台（包括高通、三星、华为、联发科等）的性能表现，并发布了一个手机的 AI 性能排行榜。从调研的结果可以发现，不同的手机平台，其中的 AI 加速芯片各不相同，具有不同的架构和编程的环境。AI benchmark 主要针对的是 CNN 应用的计算延迟，并且不考虑设备运行 DNN 的功耗。Jack 等[74] 主要衡量了跨软件栈的 CNN 的优化技术（包括权值量化、权重剪枝和通道剪枝），分析发现权值量化和通道剪枝能够很好的减少推理延迟，而权重剪枝不能。Jack 等的研究[74] 的研究没有覆盖更多的硬件架构，也没有衡量不同的优化技术对于功耗的影响。并且随着研究的发展，已经有不少工作可以实现权重剪枝的加速（例如[31]）。Merck 等[75] 研究了以 Raspberry 3 为核心计算平台的 iRobot2 机器人协同运行 Alexnet 时的功率、功耗和通信的延迟。其研究测量了机器人在靠近工作站（Station）、远离工作站和在移动的情况下通信延迟的直方图。这个工作表明，对于协同运行的应用，设备不同的移动状态会极大的影响端到端的推理延迟（通信的延迟从平均的 73ms、方差 12.2ms 变化到平均 81ms、方差 34.9ms）。Samuel 等[76] 刻画了两种部署方式：只在移动设备上和只在云端设备上移动应用使用 DNN 的推理延迟。研究发现新的设备可以

在用户可接受的范围内推理面向移动设备优化的 DNN，而老的移动设备要想推理复杂一些的 DNN，就只能采用基于云端部署的方式了。于是作者提出了一个算法 CNNSelect 来自动的选择合适的 CNN 模型，在平衡推理的延迟和准确率的前提下执行推理的任务。实验结果表明，在保证服务质量协议（SLA）的前提下 CNNSelect 能在 88.5% 的工作负载中表现的比贪心算法更好。Simone 等[77] 分析了超过 30 个不同网络结构的面向图像分类的 CNN 网络。对于每个 DNN，从多个维度（包括识别的准确率、模型的复杂性、内存使用和推理时间）分析和讨论了其性能表现并做出了详细的对比。作者也采取了两种不同计算能力的平台：装备有云端 GPU 卡的 NVIDIA Titan X Pascal 和嵌入式的 Jetson TX1 开发板。作者的工作对于应用选择合适的网络架构具有重要的指导意义。EEMBC MLMark[78] 是面向嵌入式设备设计的衡量性能和准确率的基准测试套件。它主要包括 3 个模型：Resnet-50 V1、Mobilenet-v1 和 SSD-MobileNet-v1。EEMBC MLMark 的测试集太少，只有 3 个 CNN，缺乏代表性；此外也不支持最新的模型优化的技术。

主要面向云端 DNN 基准测试和性能刻画有以下的相关工作。MLPerf Inference[79] 是一个工业级的基准测试集，专注于 DNN 模型的推理，由超过 30 个组织和 200 个机器学习的工程师参与制定完成。MLPerf Inference 制定了一系列的规则，来保证不同系统之间的可对比性。MLPerf Inference 目前（v0.5）主要由 5 个模型组成：Resnet-50、Mobilenet-v1、SSD-Resnet-34、SSD-Mobilenet-v1、GNMT。MLPerf Inference 的测试结果表明，嵌入式设备（包括智能手机）到数据中心系统其性能相差 4 个数量级。MLPerf 的主要缺点是不支持衡量功耗，而功耗是嵌入式设备非常重要的评价指标。Jussi 等[80] 研究了 CNN 模型在移动端和云端平台的计算延迟和吞吐。此外，研究人员也研究了 DNN 模型的按层的基准测试。Kim 等[81] 分析了几种不同的卷积算法的性能特征，并且衡量了在训练和推理的过程当中，DNN 每层的计算延迟。这个研究表明，DNN 模型在移动设备上推理的时候，每层的计算延迟、功耗和对应的底层的软硬件的性能指标。Karki 等[82] 提出了一个新的 DNN 基准测试套件，这个套件可以在任何支持 CUDA 和 OpenCL 的平台上运行，并且通过衡量 5 个 CNNs 网络和 2 个 RNN 的网络，作者深入的分析了这些网络在体系结构上性能特征的统计，分类展示了 DNN 不同的层的计算延迟、能量消耗和指令类型。Turner 等[83] 实现了几种常用的 DNN 模型压缩的技术（包括权重剪枝、通道剪枝和量化）并且衡量了他们的准确率、在 CPU 和 GPU 上的执行时间和内存占用。他们发现通道剪枝可以极大的减少执行时间，然而随机的细粒度的权重剪枝不能够减少执行时间。他们的工作为本文了解剪枝对推理延迟的影响提供了重要的参考。Wang 等[84] 提出了一个参数化的基准测试套件：ParaDNN。针对云端平台，PatDNN 刻画了 TPU、CPU 和 GPU 在训练 DNN 模型（包括 MLP，CNN 和 RNN）的性能表现。他们提供了 14 个主要的观察并且分析了 TPU、GPU 和 CPU 的优点和缺点。他们的工作主要集中在云端平台的 DNN 训练上，关注批处理的延迟和吞吐。[85] 提出了一个面向 DNN 训练的基准测试套件：TBD。TDB 由 8 个代表性的 DNN 模型组成，覆盖了图像分类、机器翻译、语音识别、物体检测、生成对抗网络和强化学习等任务；

TDB 使用三个主流的深度学习框架（Tensorflow、MXNet 和 CNTK）在不同的硬件配置（单 GPU、多 GPU 和多机多卡）下深度的分析了训练模型的 GPU 浮点效率等。[86] 提出了一个专门面向云端 GPU 的基准测试套件 DNNmark，DNNMark 可以分析 DNN 中不同类型的层（Convolution、Polling、Relu、Fully-Connectted、Softmax 等）使用不同的后端库（CuDNN、CUDA、CuDNN）GPU 的浮点利用率。[87] 主要关注于测试目前最顶尖的机器学习系统（包括 NVIDIA DGX-2、Amazon Web Services (AWS) P3、IBM Power System Accelerated Compute Server AC922 等）训练 CNN 和 RNN 的性能。其不仅关注单个加速设备训练的性能，同时也关注系统的可拓展性。AI Matrix[88] 是基于 Tensorflow 和 Caffe 的面线智能处理器的基准测试框架，其包含四类测试：底层测试，分层测试，完整测试，和合成测试 (StatsNet)。其中合成测试 StatsMet 是其主要的一个创新点，通过合成模型从统计的角度来模拟已有的模型，同时又有足够的灵活性以测试不同的硬件。

2.1.2　性能调优：移动端-云端协同计算

以往的移动端-云端协同计算更多的关注移动应用将应用的一部分代码上传到云端执行。通常云端需要运行移动端应用的一个备份或者是使用相同的软件环境（例如利用托管式语言在移动端和云端分别运行虚拟机）。MAUI[89] 发挥了托管式语言的优势，其首先根据运行时的统计信息刻画一个应用的性能特征，来发现应用中的热点代码，之后决定是否把一个函数调用上传到云端执行，从而最小化移动设备的能量消耗。实验表明 MAUI 对于一个人脸识别应用能够减少一个数量级的能量消耗；对于一个延迟敏感的街机游戏实现了刷新率的翻倍；对于一个基于语音的语言翻译应用，能够通过远程执行移动设备不支持的组件绕过移动设备本来不支持的限制。这几种端云协同的框架主要是以函数或者线程为单位将代码上传到云端执行。对于一些特定类型的应用，不仅能够减少延迟或者功耗，还能够在不给程序员带来大的负担的情况下，拓展移动设备的功能。然而现在的深度学习应用通常都是基于 C++ 语言，MAUI 就不适用了。CloneCloud[90] 探究了如何无缝的上传任意以函数为单位的代码到云端。具体的，CloneCloud 结合静态分析和动态的性能收集，在运行时动态的以函数为单位细粒度地划分应用，当执行计算密集代码并且在网络通讯条件良好的时候，把移动端正在执行的一个线程上传到云端，云端运行着和移动端一样的应用的“克隆”，这样就使得运行在虚拟机中的移动设备应用无需修改，就可以在运行时无缝的把其中一部分执行的代码上传到云端执行。云端执行完成后，把执行的结果和程序的状态再发送给移动端，移动端从返回的状态开始，继续执行应用。作者的实验表明，CloneCloud 可以适应各种环境下的应用切分，并且能够帮助某些应用实现最高 20× 的加速，降低移动设备的能量消耗最低到原来的 1/20。Comet[91] 比 CloneCloud 的粒度更粗，其允许直接迁移整个线程到云端计算。Comet 基于安卓的 Dalvik 虚拟机实现。Comet 利用 Dalvik 虚拟机底层的内存模型实现了分布式的共享内存。得益于新的虚拟机同步原语，Comet 只有在很少的情况下才有迁移的限制。此外当遇到网络连接失败的情况时，Comet 基于虚拟机提供的丰富的

信息，也能够很好的恢复执行。Comet 证明了其对 Google 商店的许多应用（包括图像处理、基于回合制的游戏等）都有很好的加速效果。实验表明，在 Wi-Fi 和 3G 条件下，相比 Dalvik 解释器，Comet 能够实现平均 2.88× 和 1.27× 的加速比，对某些应用最高能实现 15× 的加速。Comet 通过 Dalvik 虚拟机迁移其中的线程，然而对于深度学习应用，通常都是使用 C++ 语言，或者使用异构计算（例如 OpenCL 或者 Vulkan），其不能很好的处理这种情况。

Neurosurgeon[92] 提供了一个不同的思路。Neurosurgeon 以 DNN 的层为单位，将 DNN 模型切分成两部分部署到移动端和云端执行。具体来讲，其首先测量 DNN 每层（算子）在移动端设备和云端设备的计算延迟、传输数据的延迟和功耗。之后选择一个层切分，将这一层的数据上传到云端执行。云端执行完后续的 DNN 层，将推理结果返回给移动端。Neuralsurgeon 在 7 个典型的网络上达到了平均 3.1× 加速比，并且相比只有云端执行的方式，可以达到最高 40.7× 加速比。当考虑到能量消耗，Neurosurgeon 达到了平均节省 59.5% 的移动设备的能量消耗。Neurosurgeon 与 Caffe 集成，而不能方便的更换不同的后端，并且不能够支持移动端不同软件配置、不能衡量智能处理器、DNN 模型优化对端云协同的影响。Mao 等[93] 从无线通信的角度，总结了在 5G 通信的时代，Mobile-Edge-Compinting（MEC）所面临的机遇与挑战，包括通信和计算资源的管理和隐私敏感的 MEC 等。Surat 等[94] 提出了一个面向云端、雾端和终端设备的分布式的 DNN 推理框架 DDNN。除了能够支持在云端的推理，DDNN 还支持快速的基于局部性和地理位置的本地的协同推理。在训练网络模型时，DDNN 能够把一个网络的不同部分分别在云端和移动端进行训练。实验表明，相比传统的把原始的传感器收集到的数据传送到云端的方式，在达到相似的准确率的前提下，DDNN 本地的处理传感的数据能够把通信的开销减少 20×。Kea 等[95] 提出了基于基准测试的结果，决定移动设备获取到的传感器数据应该在哪里执行的系统。Kea 主要针对移动设备传感器产生的流数据，通过考量移动设备的计算能力、网络通信的延迟和流计算特征来最小化计算的能耗和延迟。Kea 没有考虑到不同的 DNN 模型对于准确率的影响，也没有能够衡量 DNN 模型优化技术对推理的影响。[96] 提出了基于分割的 DNN 边缘计算技术神经网络增量迁移框架 IONN。IONN 将客户的 DNN 模型划分为几个子图，并将它们一个接一个地上传到边缘服务器。当每个 DNN 子图到达时，服务器以增量方式构建 DNN 模型，从而允许客户端在上载整个 DNN 模型之前就开始执行部分 DNN。为了确定最佳的 DNN 子图和上传顺序，IONN 使用了一种新颖的基于图的算法。作者的实验表明，IONN 可以在实际的硬件配置和网络条件下显着提高查询性能。[97] 针对移动设备不断移动，需要不同的服务器连接，需要将 DNN 模型上传到云端的场景，使用了一种增量迁移神经网络（IONN）的方法。这样，当有 DNN 推理的请求，移动端就可以执行一部分，云端执行已经迁移到云端的层的部分。作者提出了一种划分的算法。算法根据迁移的收益除以迁移的开销来启发式地寻找切分点。实验结果表明作者的方法最多能够提高 55%DNN 推理的性能。Jeong 等[96] 和 Shin 等[97] 都是基于 IONN 的方式来进行分布式的推理。然而这种方式没有考虑到不同的软硬

件配置对移动端推理的影响，从而难以寻找到最优的划分方式。Fang 等[98] 专门面向移动视觉应用，开发了一个类似数据库查询语言 SQL 查询的 API，程序员能够通过写 SQL 语句的方式，将移动端 DNN 推理的任务迁移到云端执行；并设计了一个调度算法，在预测 GPU 负载的前提下，把非实时的任务打包成一个批次执行，从而减小对实时任务的干扰。Fang 等[98] 没有衡量语音识别、语言翻译等 DNN 应用对端云协同运行的影响。Zeng 等[99] 针对工业物联网的场景，提出了面向移动端的按需的协同运行 DNN 的框架 Boomerang。Boomerang 主要采用了两种技术来降低推理的延迟。第一种是 right-sizing，通过提前退出的方式减少 DNN 的计算量；第二种是 DNN partition，动态地把切分 DNN 到移动端和移动端服务器，利用服务器的异构计算资源来实现 DNN 加速推理。Zeng 等[100] 的工作表明指出，以往的移动端迁移系统都是认为云端服务器被一个移动设备独占，然而实际生产环境中都是一个云服务器要服务多个移动端设备，多个移动设备的请求就会形成竞争，从而影响服务质量。作者提出了一个调度技术来提高多租户（服务多个移动设备）的计算机视觉任务的服务质量。调度技术使用了计划-调度（Plan-Schedule）的方法。在计划阶段，算法预测未来的所有租户的工作负载，预测资源的竞争，然后调整未来任务的启动时间来减少资源竞争。在调度阶段，算法把迁移过来的任务分配给服务器。对于在 GPU 上运行的 DNN 负载，作者提出了动态打包（batching）的算法来达到最佳的服务质量。Zhang 等[101] 调研总结了移动端的信息系统（Edge Information System, EIS），包括 edge caching, edge computing 和 edge AI，这些技术在未来催生会许多激动人心的 Internet of Vehicles（IoV）应用。

也有越来越多的研究开始关注移动设备之间协同运行深度学习应用的场景。TeamNet[102] 是一个分布式移动端协同推理框架。TeamNet 假设在训练的时候，移动端的节点只学习了整个数据集中一部分的结果，每个移动端在一部分的数据上比其他移动端有更高的置信度。在推理的时候，TeamNet 基于竞争和选择算法，将 DNN 的推理任务分发到周围的移动端节点执行，之后收集结果并选择置信度最高的结果。结果表明采用 TeamNet 算法的 DNN 推理和其他实验中对比的方法相比，最多能快 53%。TeamNet 为联邦学习的训练和推理提供来有益的指导。但是 TeamNet 需要把数据广播给附近的移动端设备，只适用于物联网的场景，而不适用于智能手机这种对隐私要求比较高的场景。Funai 等[103] 指出，随着越来越多的移动设备都原生的支持（点对点）ad-hoc 通信协议，大的 ad-hoc 网络不仅能够加速移动设备之间的通信，也能够用来帮助移动设备迁移计算密集的应用。以往的计算迁移工作集中在把计算迁移到只有一跳的网络，只允许一跳极大限制了能够参与应用迁移的设备的数量。作者将应用迁移看作一个广义分配问题（General Assignment Problem），提出了一个迭代的任务分配算法：Augmented Form of Linear Bottleneck Assignment，可以尽量减少网络划分的同时优化应用在多跳网络的迁移。实验表明相比贪心算法，作者的算法能够支持更多任务的执行。Funai 等[103] 的研究说明，在未来的环境中，移动端之间的协同运行变得越来越重要。Funai 等[103] 主要集中在对网络的研究。北大的[104] 提出了一个面向

可穿戴设备的深度学习框架 DeepWear。DeepWear 主要是通过蓝牙将 DL 的任务迁移到与可穿戴式设备相连的手机上进行协同地推理。相比通过网络向云端迁移 DNN 的计算，DeepWear 在不依赖网络连接、消耗更少的能量同时，还能够更好的保护隐私。DeepWear 提供了许多技术（例如上下文敏感的迁移、模型切分和 Pipeline）来有效的利用附近的手机。同时 DeepWear 提供了对开发者友好的 API。作者基于 Android wear OS 实现了 Deepwear，实验表明与只在可穿戴设备上部署（或只在手机上部署），Deepware 能够达到 5.08× 和 23.0× 的加速比，同时节省 53.8% 到 88.5% 的能量。然而，DeepWear 只针对可穿戴设备，通过蓝牙迁移到手机上，而不能更灵活的迁移到不同的设备，例如其他的穿戴设备或者云服务器上。此外，Chatzopoulos 等[105] 更近一步，在此前 Mobile-Cloud Computing 方法的基础上，设计了基于近场通信技术的协议来支持计算迁移。Zhao 等[106] 探索了在时分复用的系统上有效的支持端云协同的服务。

2.1.3 性能调优：多计算单元异构并行推理

之前针对 DNN 模型推理的优化可以分为两个方面：（1）利用异构硬件资源和（2）优化模型。优化模型的算法包括以下的相关工作。Deep-compression[107] 提出了一系列的优化模型的方法。首先对不重要的权值进行剪枝，之后通过权值共享的方式减少权值的存储，最后通过霍夫曼编码，进一步减少权值的比特数目。实验表明能够获得 10 – 50× 的压缩率；在 Alext 上可以实现平均 3× 的推理加速。针对 DNN 模型的剪枝、量化还有非常多的研究工作（如[30]）。

DeepMon[108] 在利用手机的 GPU 进行加速做了先驱的工作。其通过对 DNN 应用按层（layer-wise）的分析发现，发现卷积层占据了推理的绝大部分时间。于是作者使用 OpenCL 和 Vulkan 实现了卷积的算法，把在手机 CPU 上的卷积计算迁移到了手机 GPU 上，实现了针对 AI 应用的异构加速。之后，越来越多的深度学习框架开始针对移动设备和嵌入式设备进行优化。其中包括 Google 提出的 TFLite[40]、小米主导开发的 MACE[41]、腾讯主导开发的 NCNN[109]、百度的 Paddle lite[110]、阿里巴巴的 MNN[16] 等。TFLite 引入了代理的概念，把要在异构设备上执行的算子组合成一个子图，然后交由异构设备的代理执行。TFLite 支持多种后端，包括通过 OpenGL 调用安卓 GPU、通过安卓系统的 NNAPI 调用 GPU 和 NPU，以及调用 Hexagon DSP。此外 TFLite 支持量化和稀疏模型的推理。MACE、NCNN、MNN 都通过 OpenCL、Vulkan 等 API 实现了对 GPU 的异构调用，并且支持把算子迁移到 GPU 执行。TVM[111] 是一个开源的深度学习编译器，它可以面向多种后端平台生成高效的可执行代码。TVM 可以自动调优生成代码，例如选择矩阵乘法分块的大小、进行循环展开、向量化和算子融合等等。TVM 也支持面向移动设备生成高效的 CPU 和 GPU（OpenCL）的代码。尽管这些框架都支持在移动设备上的异构计算执行 DNN 模型的推理，然后他们都只能顺序地执行算子而不支持异构并行地执行 DNN 模型。

目前 DNN 异构并行的研究工作主要集中在云端分布式训练场景中。随着模型越来越复杂、数据集越来越大，单机的训练已经无法满足需求，需要在分布式

环境中使用多种设备（CPU、GPU、TPU 等）进行分布式训练。分布式训练需要把 DNN 模型的算子分配到不同的设备上。传统的方法需要程序员基于经验和直觉，手工把算子映射到具体的设备上运行。Azalia 等[112] 提出了一种基于强化学习的方法，可以在训练的过程中自动的把 Tensorflow 的计算图分配到可用的计算设备上。实验表明作者的模型在 Inception-v3 训练图像分类的任务、RNN 在语言模型的任务和机器翻译的训练上比手工分配最快有 1.23× 加速比。Gao 等[113] 提出了 Post 算法，相比[112] 更进一步，由于强化学习需要不断的执行 Monte-Carlo 搜索，在没有偏好指导的情况下，搜索的过程是均匀地设置算子到设备的映射，因而还是会存在很多没有被探索的空间。Post 使用了一个新的联合学习算法，集合了最小化交叉熵和近端策略优化，来达到具有理论保证的优化结果。作者在几个常用的 DNN 训练的基准测试中表明了 Post 的性能：在相同的学习时间相比最优的结果能缩短 63.7% 的时间。Azalia 等[112] 和 Gao 等[112] 的方法都需要在训练过程中不断的迭代，设备分配的开销比较大，往往需要数分钟甚至数小时才能求得比较优的结果。而移动端受限于功耗和用户响应时间的要求，需要能够实时的计算出设备分配的结果。Demirci 等[114] 针对超算中在全局和单机的功耗约束条件下的并行任务调度问题，设计了基于分治算法的任务设备分配算法。实验表明作者的算法相比贪心算法最多能够提高 75% 的性能。但是这个方法把设备都看作是同构的，不能够很好的处理设备之间数据转换的开销。

目前针对移动设备的异构设备分配最优的工作是 MOSAIC[115]。MOSIAC 的基本观察是当前越来越多的移动设备集成了异构加速芯片（GPU、NPU 和 DSP 等）。然而对于不同 DNN 模型中的算子，不同的计算单元具有不同的性能、功耗通信开销和约束。不存在一个计算单元在所有的 DNN 模型的所有算子都能达到最优（最低延迟或者最低功耗）。因此，如何将 DNN 模型的算子分配到不同的计算单元，实现最低的延迟和功耗是一个问题。作者基于 DNN 模型分层的延迟和功耗的测量结果，考虑在不同设备之间传输数据的通信开销的前提下，使用动态规划算法将 DNN 层分配到不同的设备上。实验结果表明，MOSAIC 能够极大的降低延迟和能耗。然而，MOSAIC 不能够挖掘 DNN 模型中的算子间并行性，认为 DNN 仍然是算子顺序执行的，不能够同时调用不同的计算单元进行并行计算。μLayer[48] 针对每个算子使用多个计算单元进行加速。μLayer 发现单个计算单元的性能有限，于是他们把算子在通道维度拆分到手机的 CPU 和 GPU 上执行并行计算，并且在 CPU 是使用整数，在 GPU 上使用 16 位浮点数进行量化。实验表明 μLayer 最多能够加速执行 69.6%。我们在第 3.6节中有和 MOSAIC 以及 μLayer 进行性能对比。

2.1.3.1　小结

近年来，随着深度学习的兴起，研究人员针对深度学习模型在多种软硬件环境中进行了基准测试，并且在编程框架中做了许多性能优化，以加速 DNN 模型的推理性能。然而，当前基准测试集其涵盖的 DNN 的类型较少，针对移动端较少的考虑功耗的影响。在优化 DNN 性能推理的方面，通常只考虑将 DNN 模型

映射到一个计算设备或者计算单元上，这限制了 DNN 模型使用多种异构计算单元协同推理的能力。在多计算设备上（例如手机和云服务器），随着 5G 和云计算计算的发展，我们能够将 DNN 模型的算子映射到移动端和云端进行协同地推理；在多计算单元上（例如 CPU 和 GPU），可以将 DNN 算子映射异构计算单元上并行执行。如何刻画 DNN 在多种软硬件上的性能特征，还有许多需要探索的内容；如何自动的 DNN 负载映射到多种计算设备和计算单元以优化性能仍然是一个开放的问题。

2.2 DNN 推理的编译优化

对在软件层面 DNN 推理的优化的工作大致可以分为两类：一类是在算法层面，通过改进 DNN 模型来降低推理所需要的计算量。这类工作包括对 DNN 模型进行量化、剪枝、蒸馏、结构优化等。另一类是在系统软件层面，通过改进运行 DNN 模型的软件，包括 DNN 推理框架、算子库以及 DNN 编译器等来加速 DNN 模型的运行。本文主要从系统软件的角度介绍加速 DNN 推理的相关工作。在深度学习兴起的早期，对 DNN 推理的优化工作主要集中在对深度学习的推理框架（例如 TensorFlow、TFLite 等）的优化。注意，本文把通过直接函数调用运行 DNN 模型的称为推理框架，把先进行代码生成而后编译成二进制运行的称为 AI 编译器。在前端，推理框架以算子和张量组成的计算图为中间表示，进行传统编译器中常见的优化，例如代数变换、常量折叠、公共子表达式消除、死代码消除等。在后端，推理框架通常选择调用更优的算子实现库，例如 cuDNN 等来实现推理加速。随着 DNN 模型的和面向 AI 开发的硬件加速器的发展，人们发现单纯优化 DNN 框架已经不能够充分的发挥底层硬件性能，于是提出了多种深度学习编译器来简化编程并进行多种编译优化。算子融合和单算子的性能优化是其中的热点研究话题。本文也主要关注这两方面相关工作的发展脉络与研究进展。

2.2.1 算子融合

2.2.1.1 基于模式匹配的算子融合

TensorFlow 和 PyTorch 等 DNN 框架在早期通过手工重写新算子的方式来进行算子融合。例如要融合“Conv2D”和“BatchNorm”两个算子，程序员需要在框架中注册并实现一个新的“FusedConv2DBatchNorm”算子。DNN 计算图的优化器在扫描到“Conv2D”和“BatchNorm”后，就将这两个算子替换为“FusedConv2DBatchNorm”。这种方式的工作量大，可拓展性很低，只能够针对最常用的算子组合进行优化。HFTA[116] 在模型训练阶段将来自于不同模型没有依赖关系的算子合并为已经定义的算子，从而提高硬件的利用率。TASO[46] 通过自动生成多个算子组成的子图的等价替换实现计算图的优化。TASO 主要支持相同类型的无数据依赖的算子融合。它先使用连接（Concatenation）操作将输入数据合并，经过融合后的算子计算后，再将各自的结果切分开（Split）。

2.2.1.2　基于规则的算子融合

算子融合通过增加数据复用、减少片外的内存访问从而带来显著的性能提升。TensorFlow 的 XLA[50] 是进行 DNN 推理编译优化的先行者，其将上层的 DNN 计算图先变换为高层次优化器（High Level Optimizer，HLO）的中间表示（Intermediate Representation，IR），然后进行算子融合等编译优化，面向 CPU、GPU 和 TPU 生成二进制代码。XLA 的算子融合基于规则的方法，将 HLO 的 IR 进行分类，将计算模式简单计算量低的算子合并为一个算子，生成一个计算核（kernel）。

TVM[51] 提供了 *compute_inline* 的调度原语，可以将两个循环边界相同的计算合并到一个循环中，从而实现算子融合。在 TVM Relay[117] 按照表达式的复杂度算子分为七个级别，定义了不同级别算子之间融合的规则。

DNNFusion[118] 根据算子输入和输出张量之间的元素映射关系，将算子分为五种类型：一对一（One-to-One）、一对多（One-to-Many）、多对多（Many-to-Many）、重组（Reorganize）和洗牌（Shuffle）。DNNFusion 定义了不同类型的算子之间组合的规则，将不同类型的算子进行融合。DNNFusion 还提供了基于算子数学性质的等价算子替换、以及优化数据搬运算子等。DNNFusion 主要面向移动平台优化。

DeepCuts[119] 同时考虑算子的计算特征与 GPU 的参数来进行算子融合。DeepCuts 提出了一个基于 GPU 简单的性能代价模型（Cost Model），预测算子融合后的代码是否会有性能提升。算法主要从两个方面优化性能：算子融合和 kernel 参数搜索。DeepCuts 从共享内存层（Shared Memory Level）和寄存器层考虑算子间优化的机会。之后根据代价模型进行剪枝，从而快速筛选出性能较优的生成 kernel 的参数。

之前的工作主要关注有数据依赖关系算子的融合（通常称之为垂直融合）。Rammer[62] 提供了一个新的算子融合的角度：将互相之间没有数据依赖关系的算子进行融合（通常称之为水平融合），以挖掘算子之间的并行性，充分利用 GPU 和智能处理器多个计算部件并行计算的能力。在 DNN 计算图层面，Rammer 以 *rTask* 作为调度单位，来统一算子内（单个算子）和算子间（多个无数据依赖关系的算子）的调度。在硬件层面，Rammer 抽象出虚拟并行计算设备（Virtual parallel device，*vDevice*）的概念，作为调度单位。最后，Rammer 基于 *rTask* 在 *vDevice* 上性能测试的结果进行最终的调度与代码生成。与厂商优化的 TensorRT 相比，Rammer 能够获得最高 3.1× 的加速比。

2.2.1.3　基于依赖分析的算子融合

Astitch[120] 将算子分为访存密集型(Memory-intensive)和计算密集型(Compute-intensive) 两种类型。其贪心的融合访存密集型算子。相较于以往的工作，Astitch 有两方面的创新：第一：Astitch 可以在多个内存的层级（包括寄存器、共享内存和全局内存）融合规约算子（例如 Reduce_sum、Reduce_max 等），从而发掘更

多在片上内存进行数据复用的机会。第二：Astitch 可以通过动态的线程绑定以适应张量的动态形状，从而可以在运行时对不同形状的张量进行优化。

Apollo[121] 提出了一个基于图划分的方法，将 DNN 的模型切分为多个子图，然后在子图内部寻找算子融合机会。相较于之前的工作，其创新点在于将复杂的算子分解为多个基础算子，而后在基础算子之上进行融合。Apollo 结合了基于规则和 Polyhedral 循环融合的编译分析方法，同样支持规约算子的合并。

2.2.1.4 传统编译中的循环融合

印度科学与理工学院的研究人员提出了一个基于动态规划的算法，结合精确的代价模型来融合图像处理管线中的操作[122]。其代价模型能够更精确的预测程序在不同分块分块下的代价，从而在数据局部性和并行性之间取得性能更优的折衷方案。其他面向通用循环融合优化的工作还包括[123–125] 等。这些工作更多面向的是针对 CPU 并行计算通用程序的循环融合。

2.2.2 深度学习编译器与单算子编译优化

随着深度学习的发展，研究人员研发出多个深度学习编译器，包括 TensorFlow XLA[50]、TACO[126]、Tiramisu[127]、TVM[128]、Tensor Comprehension[129]、nGraph[52]、Glow[130]、TensorRT[131]、Exo[132]、MLIR[133] 等。TVM 是其中具有代表性的研究工作。它采用了 Halide[134] 中计算与调度分离的思想，首先基于张量表达式（Tensor Expression）来描述算子所表达的具体的数学运算，之后基于多种调度原语（Schedule Primitives）对张量表达式进行调度，最后根据调度的结果进行代码生成。本文研究工作基于这些编译器，利用了这些编译器提供的许多基础设施。

编译器需要为编程框架提供面向不同后端硬件高性能的算子实现。通常一个算子有多种不同的逻辑等价的实现，但是最终的性能却可能差异巨大。为了能够面向不同后端、多样的算子提供性能优异的算子实现，研究人员提出了许多不同优化单算子性能的方法。AutoTVM[63] 是 TVM 编译器中为算子自动提供高性能实现的模块。它首先由程序员定义调度的模板，设置模板中的可变参数(例如循环分块的大小、是否交换计算轴等)，之后通过机器学习的方法（例如 XGBoost[135]）结合实际的性能测试结果，来选择一组最优的调度。FlexTensor[64] 相较于 AutoTVM 提升了（1）机器学习搜索调度的算法，(2) 对于最常见的算子，无需程序员写调度模板。Ansor[136] 则构建了一个代价模型，对于所有的张量表达式无需程序员写调度模板，相较于 AutoTVM 具有更大的搜索空间。Roller[65] 在 Ansor 的基础上，将调度的选择与硬件的特征对齐，极大地缩减了需要搜索的空间，最终只需要分钟级的时间就能够达到 Ansor 搜索几个小时的效果。这些研究工作与本文工作正交，本文利用这些基础设施来生成代码。

2.2.2.1　小结

如何让深度学习编译器为 DNN 模型生成性能更优的智能处理器代码是最近几年的热点话题。研究人员从单算子性能优化、算子融合、计算图优化以及 DNN 算法与编译器协同优化等多个方面展开了研究。其中，算子融合是提升数据局部性、发挥智能处理器计算峰值性能的重要优化手段。然而，当前的算子融合只考虑在计算图的局部几个算子，不能从全局的角度分析 DNN 模型架构的特点。此外，当前算子融合的工作也主要集中在张量元素依赖关系简单的算子的融合，不能够打破算子之间的界限，考虑更多的数据复用和指令调度的机会。因此，如何全局分析 DNN 模型、打破算子界限，生成性能更优的代码成为一个亟待解决的问题。

2.3　DNN 推理的数据预处理优化

2.3.1　深度学习应用中的数据预处理瓶颈分析

随着面向深度学习软硬件的发展，DNN 模型的推理速度已经得到了极大的提升。这时，研究人员发现，在某些深度学习应用中，DNN 执行的延迟不再成为整个系统的瓶颈。研究人员发现，对数据的预处理（例如解码、重新调整大小等）操作可能会成为许多计算机视觉应用的瓶颈。SMOL[67] 是一个专门针对图像预处理的框架，其主要三种方式优化数据预处理的过程：(1) 让数据预处理和 DNN 的执行形成流水线；(2) 精细的管理内存和 CPU 线程，以达到更高的处理效率；(3) 将数据直接放置到 DNN 模型所在的计算单元上。Intel 在实际的模型部署中发现，OpenCV 进行图像编解码以及图像预处理的部分会成为整个 DNN 中的瓶颈[66]。PLUMBER[137] 被开发出以发现训练过程中数据处理的流水线中哪里有性能瓶颈，并通过调节分配给数据预处理程序的硬件资源来消除性能瓶颈。慕尼黑工业大学的研究人员从数据存储、内存缓存、压缩和 CPU 多核并行等角度分析了多种因素对数据预处理的影响[68]，他们发现在以上的几个方面都有可能成为深度学习数据预处理过程中的瓶颈。

2.3.2　深度学习中的图像处理

目前，对 DNN 中图像的处理主要依赖于 OpenCV[69]、Python Image Library (PIL) [138] 等图像处理库。这些库包含了许多经典的图像处理算法的手工实现，主要针对 CPU 和 GPU 进行优化。然而，越来越多的 DNN 模型在定制的智能处理器上训练和部署，于是，如何在智能处理器上运行图像预处理和后处理算法成为一个关键的问题。像 OpenCV 这样的库通常由人工根据经验调优，而 Halide[134] 针对图像处理流水线中并行度、局部性和重计算（Redundant Recomputation）之间的折衷，提出了计算（Algorithm）和调度（Schedule）分离的编译器设计思想。程序员先用 Halide DSL（Domain Specific Language）描述图像处理的计算，之后编译器基于机器学习的算法在调度空间中进行搜索。程序员只需要写几个小时

Halide 程序，就能够达到甚至超过原来手工优化几周的程序的性能。Halide 的设计思想影响了后来诸多编译器的设计，包括 TVM[128]、EXO[132] 等。

Kornia[139] 是一个基于 PyTorch[140] 开源计算机视觉库。由于其基于 PyTorch 的算子实现图像操作，相比于 OpenCV 等传统的库，它的特点是可以微分，因而可以集成到 DNN 模型中参与训练模型。它由几个模块组成，包括数据增强、颜色空间变换、图形特征检测、滤波、几何变换等。与 Kornia 类似的还有基于 TensorFlow 的 TensorFlow.image[141]。TorchVision[142] 是 Torch 框架中的机器视觉模块，提供了一系列图像处理算法。其利用 PyTorch 的算子实现，因而可以和 PyTorch 框架紧密的结合在一起，完成对图像的预处理功能。TorchIO[143] 和 Kornia 类似，它主要面向深度学习医学图像（例如核磁共振成像，Magnetic Resonance Imaging），基于 PyTorch 构建。与 TorchIO 类似的还有 NiftyNet[144] 和 DLTK[145]，这两个框架都是基于 TensorFlow 实现。

CV-CUDA[146] 是 NVIDIA 于 2022 年 GPU 技术大会（GPU Technical Conference）上发布的用于构建端到端的计算机视觉和图形处理管线的开源库。CV-CUDA 主要用于加速除 DNN 模型以外预处理和后处理部分的计算任务，包括重新光照、模糊背景、超分辨率等图像和视频处理任务。与 CV-CUDA 类似的库还包括 Intel 公司开发的 oneAPI Image Processing Library（oneIPL）[147]。

以上的相关工作目前主要只支持在 CPU 和 GPU 上运行。OpenVX[148] 是由 KHRONOS GROUPS 提出了一个开源的、跨平台的计算机视觉加速库。它和深度学习框架采用了类似的编程模型，将对图形图像的一个操作抽象为算子，算子之间可以构建为计算图来表达一系列复杂的操作。OpenVX 支持图级别的优化，并且可以被部署到多种计算平台。目前 OpenVX 还在早期的开发阶段。

2.3.3 智能处理器运行非 AI 应用

尽管智能处理器设计之初的目的是用来对深度学习应用进行加速，拓展智能处理器的应用范围也是研究人员关心的热点问题。来自加利福尼亚大学河滨分校的研究人员提出了 GPTPU[149]，一个开源的框架，可以让用户使用 TPU 来执行通用计算任务。GPTPU 提供了类似于 CUDA 和 OpenCL 类似的编程接口，以及一个运行时以支持在低精度和有限编程接口的 TPU 上运行一些通用计算任务，包括高斯模糊（图像处理）、矩阵乘、PageRank（机器学习）等。其特点是对数值进行适当的缩放，最终能够以很小的精度损失得到最高 2.46× 的加速比。后来，他们又提出了 TCUDB[150]，使用张量计算单元（Tensor Core Unit，TCU）来加速数据库查询的任务，与基于 GPU 的查询引擎相比，最高能够获得 288× 的加速比。利用 TCU 来加速非 AI 应用的还包括伊利诺伊大学厄巴纳-香槟分校的研究人员，他们利用 NVIDIA GPU 中的 TCU 来加速规约（Reduction）和扫描（Scan）运算，相比于基准测试能够获得一个数量级（对于规约运算最多 100×，对于扫描运算最多 3×）的性能提升[151]。此外，TCstencil[152] 探索了使用 TCU 来加速在高性能计算中的 stencil 运算。TCstencil 将 stencil 运算重新分解为了一

系列的规约和累加操作，以利用 TCU 的矩阵乘指令。此外，TCU 还被优化以加速对图像的卷积运算[153]。

2.3.4　小结

目前，研究人员开发了一些库以便在 CPU 和 GPU 上进行图像等深度学习数据的预处理。由于 DNN 模型的执行延迟随着对智能处理器和软件栈的不断优化越来越低，研究发现对数据的预处理可能成为深度学习的瓶颈。然而，目前专门针对智能处理器进行数据预处理的相关研究还非常少。如何在智能处理器上高效地进行快速的数据处理，已经成为一个关键的挑战。更进一步的，未来如果能够将更多在 CPU 上执行的应用迁移到智能处理器上执行，进一步的释放智能处理器的应用潜能，将会是一个非常有意义研究课题。

第 3 章　深度学习模型的性能分析与自动调优

3.1　相关背景与研究动机

越来越多的移动应用开始使用 DNN 模型，但是，DNN 模型的规模也越来越大，所需要的计算资源越来越多。北大和微软亚洲研究院发布的论文[71] 表明，占据了 Google 应用商店 11.9% 下载量的顶级应用都在使用深度学习（Deep Learning，DL)。在 3 个月的调研时间（2018 年 6 月到 9 月)，使用深度学习应用的数量增加了 27%，并且其中 81% 的应用以 DL 作为其核心功能。然而，考虑到功耗和用户响应时间的限制，移动设备上的 DNN 模型比人们预期的要轻量，其内存使用的中位数为 2.47MB，推理计算量为 10MFLOPS。尽管有非常多的 DNN 模型被开发出来，但是受限于移动设备的处理能力，那些复杂的模型并没有被实际的部署到移动设备，这样就大大的限制了 DNN 模型在移动设备中的应用。并且根据 Hestness[23] 等的估计，未来深度学习的模型的准确率要想超过人类的水平，数据集需要增长 33 – 971×, DNN 模型将需要增长 6.6 – 456×。这无疑会给 DNN 模型在移动设备上的部署带来巨大的挑战。

尽管已经有许多针对 DNN 模型推理的优化技术被开发出来，目前，大多数部署在移动设备的 AI 应用仍然是通过请求-回应（request-response）的方式为用户提供服务；与此同时，芯片制造工艺的进步和面向移动设备的 AI 芯片的集成，使得移动端设备的计算性能越来越强。以 DNN 模型为核心的 AI 应用也逐渐开始向移动端迁移。例如，华为麒麟 970 芯片集成了一个专用的 AI 加速单元 Neural Processing Unit（NPU)。这样，搭载了麒麟 970 的移动设备既可以基于云端也可以基于移动端来进行 AI 任务的处理。Gartner[154] 预测到 2022 年，80% 的智能手机将会具有 AI 的处理能力。这样就给移动设备的用户带来了强大的计算资源。一方面，随着网络基础设施的建设，在云端执行推理任务变得越来越方便；另一方面，随着移动设备的计算能力越来越强，直接在移动设备上执行推理也成为了可行的方案。更进一步的，在移动端和云端协同运行 DNN 模型也成为了具有前景的解决方案。然而，云端服务器和移动端设备在 DNN 推理的性能和能量消耗差异非常大。因此，将 DNN 协同地部署到云端和移动端是一个巨大的挑战。具体来说，当一个 DNN 模型完全被部署在云端进行推理，那么其计算的延迟是最小化的，但是所有的输入数据都需要被传输到云端，因此会引入额外的网络传输开销。当一个 DNN 模型完全在本地处理，那么就不需要网络传输，但是计算的延迟将会增加。以一个具有代表性的移动 ARM CPU 高通 Snapdragon 625 为例，对于一个小的 5 层的 MLP 模型，其推理的延迟是 136ms，因此从用户的角度来说是可以接受的；但是对于一个大的 DNN 模型，例如包含了 50 层 ResNet-50，其推理的延迟是 616ms，这样高的延迟会极大的影响用户的体验。因此，ResNet-50 就更倾向于放到云端处理，这样就把端到端的延迟降低到了 217ms (包括了数据传输的开销)。这个挑战也在 Berkely 发布的《the Berkeley view of

system challenges for AI» [155] 中作为“cloud-edge system”的一部分被讨论过。研究者们也提出了一些技术来支持移动端和云端的协同处理。Neurosurgeon[92] 是代表性工作之一。Neurosurgeon 把 DNN 模型按照层切开，分别部署到移动端和云端来执行。Neurosurgeon 证明了端云协同既可以减少推理的延迟，又能够减少移动端的功耗，而这两点对于用户体验都是十分重要的。然而，决定最优的划分策略依然是很难的问题，端到端的执行延迟取决于 DNN 模型负载、负责执行 DNN 模型的框架、需要在云端和移动端传输的数据以及网络的带宽和延迟。更严重的问题是，移动平台的种类非常多，它们之间的差异（包括硬件参数、厂商提供的库等）也非常大。硬件之间的差异性使得人们更难以找到一个最优的部署策略。

本章提出了一个 DNN 调优的引擎 DNNTune，它可以帮助用户来分析 DNN 模型在不同硬件平台上的性能和功耗，并进行自动的性能调优。DNNTune 支持许多不同的硬件和软件平台，可以为用户提供每一个算子的延迟和性能特征，并进一步的根据软硬件的配置，自动发现端云协同最优的切分点。最核心的挑战来自于软件框架的多样性、DNN 模型的特征以及软件平台的多样性。第一，要为用户提供每个算子的性能分析，需要为多种软件框架自动地进行代码插桩，来监控每个算子的执行；第二，一些 DNN 模型包含一些运行时间很短的算子，如何精确的获得这些算子的性能特征是一个问题；第三，不同的硬件平台通常提供不同的库来供深度学习框架调用，因而需要将这些不同的库集成进多种软件框架中。

为了能够解决以上的挑战，DNNTune 提供了三个接口可以让用户来指定具体的配置。例如，如何来生成进行调优的模型，具体的处理平台是什么（包括硬件平台和软件框架，以及 DNN 的库）和要调优的目标是什么（延迟或者功耗）。第一步，根据用户指定的配置信息，DNNTune 会为用户选择的 DNN 框架生成所对应格式的 DNN 模型，集成用户指定的库到对应的 DNN 框架中，在 DNN 框架中插入监控每个算子执行信息的代码。第二步，DNNTune 在移动平台和云端平台分别调用指定的框架来执行 DNN 模型，收集每层的执行信息。最后，DNNTune 框架将 DNN 模型在所有可能的切分点切开，计算在每个切分点端到端的执行延迟和功耗，根据用户目标选择最优的部署策略。

本章的工作主要有以下的贡献：

- 提出了一个 DNN 调优框架 DNNTune，能够为多种不同的 DNN 软件框架、DNN 模型和硬件平台提供按层的性能特征。DNNTune 能够为特定的目标发现最优的端云协同部署策略，只需要让 DNN 模型分别在移动端和云端运行两次即可。
- 提出了一个动态的跨平台按层的性能收集机制。DNNTune 能够自动的将特定的 DNN 库集成到软件框架中，并自动的进行代码插桩，来监控算子的执行，并动态调整每个算子的执行时间。
- 选择了多种不同类型的 DNN 模型（包括 CNN、LSTM 和 MLP）和三个

移动端设备（包括低端的到高端的智能手机，既包括 CPU 也包括 GPU）来衡量 DNNTune 。实验结果表明，DNNTune 能够达到最高 20% 的性能提升和节省 15% 的能耗。

- 提出了 DNN 算子异构并行框架，可以自动地将 DNN 的算子映射到移动端设备的多个计算单元上，异构并行的执行 DNN 模型。与当前最优工作对比，可以最多减少 41.8% 的延迟。

3.2　DNNTune 框架设计与实现

在本节首先讨论构建 DNN 调优框架的挑战，然后展示了 DNNTune 的总体设计与具体实现，之后展示了三个典型的例子。

如图 3-1所示为 DNNTune 的整体架构，其读取用户的配置信息，然后生成性能和功耗的报告，同时生成一个在用户给定目标下最优的部署策略。DNNTune 包含两个模块：配置界面和基准测试引擎（benchmarking engine）。用户可以在配置界面声明要分析的 DNN 模型，处理平台（包括硬件平台、软件框架和 DNN 的库）、以及端云协同部署 DNN 模型要优化的目标（性能或功耗）。

基准测试引擎是 DNNTune 的核心模块。如图 3-1是 DNNTune 的核心模块。其中上面的三个模块用来将 DNN 模型转换为合适的格式（“Model Generator”），生成适当的硬件性能收集器（“Platform Profiler Generator”），并且为特定处理框架生成控制的参数（“Framework Controller”）。之后下面的三个分析模块根据上面收集到的软硬件信息来发现最优的部署策略。

配置界面（Configuration Interface）。配置界面可以让用户来配置 DNN 模型、处理平台和分析目标。如图 3-2展示了一个例子。具体而言，在 DNN 模型（Model）部分，用户可以指定输入（TensorFlow 或者 Caffe）模型，以及路径、输入的名称和形状等。如果要测量量化模型，可以设置量化的数据类型（例如 INT8）。在执行（Processing）部分，用户可以设置如下的参数：

- 硬件平台，包括移动端和云端的 CPU、GPU 或者智能处理器，CPU 的线程数，以及移动平台上的大小核心；
- 软件配置，包括处理的框架和 DNN 库。如图 3-2指定了两个用来调优的框架（TensorFlow 和 MACE）和两个库（Eigen 和 NNPACK）。
- 性能收集的配置，包括使用哪个性能收集工具（例如 Snapdragon profiler[156] 和 Simpleperf[157]），是否要收集功耗（默认只收集延迟）以及一些软件和硬件事件。

目标（Object）部分，用户可以设置自动调优的目标：延迟优先或者功耗优先。

模型生成器（Model Generator）。模型生成器用来把 Caffe 或者 TensorFlow 的模型转成用户指定框架需要的格式。本文选择 Caffe 和 TensorFlow 作为 DNNTune 的输入，是因为这两个框架已经有大量的模型可供使用。模型生成器中集成了

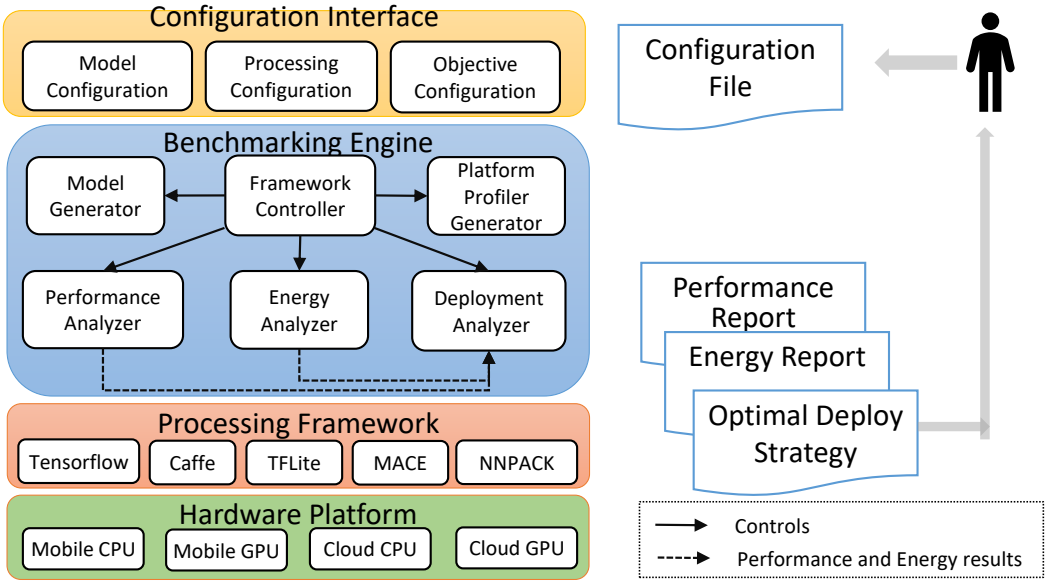


图 3-1 DNNTune 框架

Figure 3-1 The DNNTune framework overview.

```
Model
name: mobilenet-v2
file: path/to/mobilenet-v2.pb
input_name: input
input_shape:[1,224,224,3]
output_name: mobilenetv2/predictions
data_type: float

Objective
objective: energy-first

Processing
Processing_unit: mobile_CPU, cloud_CPU
CPU_threads: 4
CPU_affinity: big+little
framework: tensorflow, MACE
DNN_library: eigen, NNPACK
profiling_tools: Snapdragon_profiler,
Simpleperf
energy_profiling: true
profiling_events: CPUUtilization,
Mem_footprint, LLCmiss
```

图 3-2 配置界面的例子

Figure 3-2 An example for the configuration interface

许多格式转换工具，根据用户指定的输入模型格式可以自动选择合适的转换工具转换模型到要运行的框架上。比如说，当用户指定输入模型为 TensorFlow 格式的（Protocol Buffer，.pb 格式），要运行的框架为 MACE，DNNTune 自动调用 TensorFlow-to-MACE 的转换工具来为 MACE 生成对应的模型格式。

框架控制器（Framework Controller）。为了能够适配多种多样的软硬件平台，框架控制器是 DNNTune 中的核心模块。框架控制器用来把用户提供的配置界面参数转换为框架的参数，把特定的 DNN 库集成进 DNN 框架中，以及在 DNN 框架中插桩。(1) 配置。框架控制器根据运行的框架把用户配置的参数转换为对应的框架参数。具体而言，它会抽取与框架相关的配置，包括计算单元、CPU 线程数量、CPU 亲和性等等。这些配置会被转换为运行框架的参数。例如，对于 MACE，这些参数分别是 *device, omp_num_threads, cpu_affinity_policy*。(2) DNN

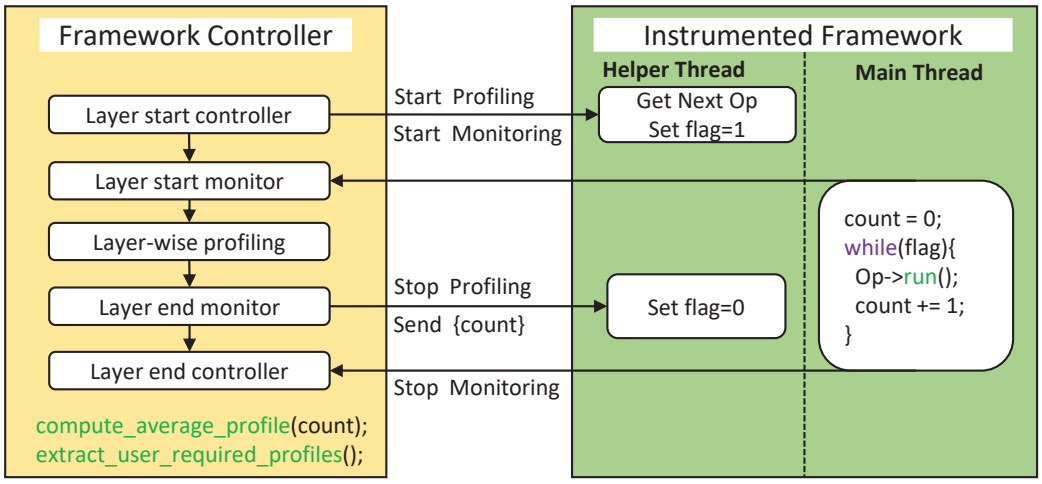


图 3-3 DNNTune 中按层的插桩控制性能收集的机制

Figure 3-3 The instrumentation for layer-wise profiling in DNNTune

库。框架控制器支持在不同的 DNN 库之间切换。比如说，TensorFlow 和 TFLite 都使用 Eigen 作为默认的 DNN 框架。但是 Eigen 的性能不够优秀。因此，这些框架可以让用户在编译的时候选择后端的 DNN 库，包括 MKLDNN、CuDNN、NNPACK 等。因此，DNNTune 首先会预先编译好多个版本的 DNN 框架，每个 DNN 库一个版本。在调优的时候，框架控制器根据用户配置选择对应的版本。(3) 插桩。框架控制器会把插桩的代码加入到运行时框架中，来监控 DNN 模型每一层的执行。其中的挑战来自于那些运行时间较短的层，其运行时间甚至低于典型的性能收集工具的采样时间间隔。例如，Snapdragon Profiler 采样工具每 50 毫秒收集一次系统功耗，但是在 MobileNet-V2 的网络中“bottleneck_3_2”层只运行 5 毫秒。为了能够获得“bottleneck_3_2”的功耗，这一层至少需要被反复的执行 10 次，才能达到采样工具的一个时间间隔。因此，本文提出了如图 3-3所示的性能收集机制，左边为框架控制器，右边为已经插桩的 DNN 框架。具体而言，在一个层开始执行之前，“Layer Start Controller”发送一个信号到已经插桩的 DNN 框架。另外，DNNTune 引入了一个帮助线程（Helper Thread）等待“Layer Start Controller”的信号。当它收到信号，共享变量 *nlayer_flag* 被设置成 1，表明这一层可以开始执行了。被插桩的 DNN 框架的主线程会一直被阻塞，直到 *nlayer_flag* 被设置为 1。之后，DNN 框架会给框架控制器反馈一个信号，表明这个层开始执行了。然后这个层会被重复执行多次。与此同时，框架控制器会收集这层的性能数据。当执行时间达到所需要的时间间隔时，“Time Interval Notifier”会发送一个信号给被插桩的 DNN 框架，*nlayer_flag* 标志又会被帮助线程设置为 0，表示这一层的执行终止。最后，这一层执行结束的信号以及执行的次数信息会被发送到框架控制器，并调用 *compute_average_profile* 函数来计算平均执行时间。(4) 自动插桩。DNN 框架大多是数据流模型，通过类似于 *op->run* 的方式调用 DNN 中的算子（层）。DNNTune 可以让用户来设置文件名称和函数名称，然后框架自动寻找 *op-run* 的函数并插入插桩代码。图 3-4以 MACE 作为例子描述了自动

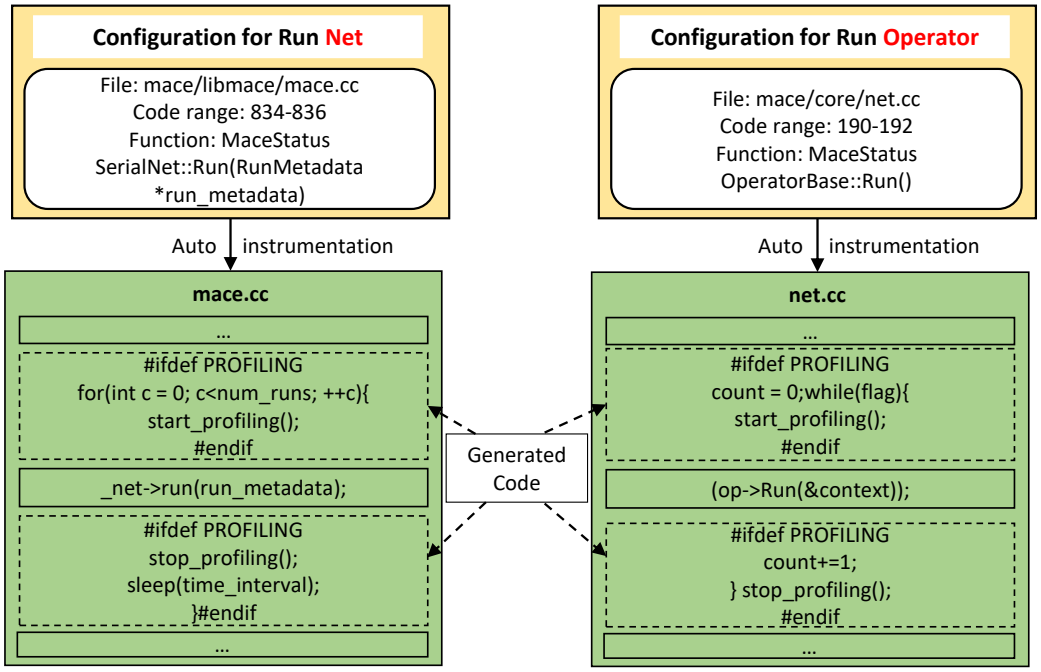


图 3-4 自动插桩的机制

Figure 3-4 Mechanism for auto instrumentation.

插桩框架是如何工作的。上方白色的代码展示了用户的配置，下方绿色的代码展示了生成的插桩代码。

性能和功耗分析器（Performance and Energy Analyzer）。性能和功耗分析器用来分析基准测试的结果，并把结果发送给部署分析器。性能分析器分析与每层的 GFLOPS、LLC miss、以及每指令周期数（Instructions Per Cycles，IPC）。功耗分析器根据算子执行延迟和采样的功率相乘来计算算子的能耗。

部署分析器（Deployment Analyzer）。部署分析器基于以下的步骤来决定最优的划分策略：(1)运行和进行性能收集：对于一个给定的 DNN 模型 d，DNNTune 首先分别在移动端和云端都运行一遍模型并记录每层的执行数据，分别记为 d_m 和 d_c。

（2）计算每个切分点的性能数据。之后，DNNTune 在所有可能的切分点把模型分为两部分，得到一个所有被切分模型的集合 P，其中的每个元素对应一个切分点 p。对于每个 $m \in P$，分析器自动地在切分点 p 切开，记第一部分为 $m1$，第二部分为 $m2$。$m1$ 和 $m2$ 分别在移动端和云端执行，它们的执行时间和功耗可以从 d_c 和 d_m 中计算得出，分别记为 T_m、T_c、E_m、E_c。

对于无线通信，本文使用和 Kang[92] 一样的设置，使用 TestMyNet[158] 来测量在 3G、LET 和 WiFi 条件下端到端的延迟。使用如下的公式来计算通信开销：

$$T_t = T_{ttl} + v_p/b \tag{3-1}$$

$$E_t = w * T_t \tag{3-2}$$

其中 T_{ttl} 是从移动端到云端网络一个来回（round-trip）的延迟，v_p 是从移动端传输到云端的数据量 v_p，b 是网络带宽，w 是通信传输模块（例如 WiFi）的功耗。因此，在切分点 p 端到端的传输延迟 T^p 和功耗 E^p 可以用下面的公式来计算：

$$T^p = T_m + T_t + T_c \tag{3-3}$$

$$E^p = E_m + E_t \tag{3-4}$$

本文只考虑移动端的能耗而忽略云端的功耗。

（3）遍历。最后，DNNTune 遍历集合 P 中所有的切分点，寻找一个能够最小化用户目标的切分点。

移动平台性能收集器。本文使用了两个工具：SimplePerf[157] 和 Snapdragon profiler（骁龙性能收集器）[156] 来实现移动平台的性能收集器。SimplePerf 是安卓（Android）系统的原生的性能收集工具，提供了命令行的接口，可以提供许多和 Linux 的 perf 工具相似的功能，并加入了一些 Android 特定的功能提升。Snapdragon profiler 是高通公司开发的面向高通骁龙系列移动片上系统（System on Chip，SoC）的性能收集工具，提供了一个图形界面来分析 CPU、GPU、DSP、内存、功耗和网络等性能信息。

3.3　性能收集框架的实验设置

本文考虑三种主要类型的 DNN 模型，它们的参数在表 3-1中。

- **卷积神经网络（Convolutional Neural Networks，CNNs）：**本文衡量了九种典型的 CNN 网络: 四个传统的 CNN 模型（Inception-V1[2], Inception-V3[159], Vgg16[1] 和 ResNet-50[3]), 四个面向移动端优化的模型(MobileNet-V1[33], MobileNet-V2[34], SqueezeNet-V11[35], ShuffleNet-V2 0.5×[160])和 (DeepLab-V3[161] with MobileNet-V2 as network backbones). 前面八个模型都是用来做图像分类的，在 ImageNet 数据集[162] 上做了预训练。DeepLab-V3 做图像语义分割，在 pascal-VOC 数据集上[163] 做了预训练。
- **循环神经网络（Recurrent Neural Network，RNNs）：**RNN 可以处理变长的数据，在自然语言处理、语音识别等领域得到了非常广泛的应用。使用最多的 RNN 是长短期记忆网络模型（Long Short Term Memory Networks，LSTMs）。本工作使用了两个 LSTM 模型，一个是 rnn_ptb_small，一个小的两层 LSTM 网络[164]，每层有 200 个隐藏单元，在 Penn Tree Bank（PTB）[165] 数据集上训练。另一个是 DeepSpeech[166]，它有六层，三个全连接层。每个 LSTM 的层有 4096 个隐藏单元，批处理的大小是 16。
- **多层感知机（Multilayer Perceptrons，MLPs）：**MLP 模型主要是用来做分类任务，其主要由一连串的全连接层组成。本文测试两个模型：一个五层的小 MLP 模型[167]，用于在 MNIST 数据集上的手写数字识别；一个大的感知模型 Transformer[9], 用于自然语言理解等任务。

表 3-1 DNN 模型参数

Table 3-1 DNN specifications

模型名称	输入	输出	层数	参数数量	FLOPs
rnn_ptb_small	[20, 200]	[20,10000]	2	2.65M	14.8M
DeepSpeech	[16, 494]	[16,29]	6	29M	464M
mnist_mlp	[784]	[10]	5	13.9M	13.9M
Transformer	[1,7]	[1,14]	12	49M	-
Inception-V1	[224,224,3]	[1000]	22	6.79M	3.19G
Incetpion-V3	[299,299,3]	[1000]	48	22.75M	6G
ResNet-50	[224,224,3]	[1000]	50	25.6M	3.8G
Vgg16	[224,224,3]	[1000]	16	138M	16G
MobileNet-V1	[224,224,3]	[1000]	15	4.2M	576M
MobileNet V2	[224,224,3]	[1000]	20	3.4M	300M
SqueezeNet-V11	[227,227,3]	[1000]	15	1.2M	360M
ShuffleNet-V2 0.5×	[224,224,3]	[1000]	15	1.4M	41M
DeepLab-V3	[513,513,3]	[65,65,21]	20	2.1M	17.8G

表 3-2 移动平台的详细参数

Table 3-2 Device specifications for edge platforms

设备	OnePlus 5T		OnePlus 3		Redmi Note 4X	
SoC	Qualcomm Snapdragon 835		Qualcomm Snapdragon 820		Qualcomm Snapdragon 625	
工艺制程	10 nm		14 nm		14 nm	
标记	Mobile A-CPU	Mobile A-GPU	Mobile B-CPU	Mobile B-GPU	Mobile C-CPU	Mobile C-GPU
平台	Kryo 280	Adreno 540	Kryo [168]	Adreno 530	Cortex A53	Adreno 506
核心数	4+4	256	2+2	256	8	96
L1 缓存	64 I+32 I KB | 64 D +32 D KB	-	32 I + 32 D KB	-	32 KB	-
L2 缓存	2+1 MB	1024 KB	1 MB+512 KB	1024 KB	1 MB	128 + 8 KB
频率	4×2.45+4×1.85 GHz	710 MHz	2×2.15+2×1.59 GHz	624 MHz	2.0 GHz	650 MHz
内存	8GB LPDDR4X		6GB LPDDR4		3GB LPDDR3	
操作系统	Android 8.1		Android 8.1		Android 7.0	

在云端平台，使用 Intel Xeon E5-2620 v4 CPU 和一个 NVIDIA Titan V100 GPU。运行的操作系统是 Ubuntu 16.04。在移动端平台，本实验衡量了三个不同性能等级的智能手机，囊括了从低端平台到高端平台。包括一加 5T（OnePlust 5T）[169]，一加 3（OnePlus 3）[170] 和红米（Redmi Note 4X）[171]。其中一加 5T（标记为 Mobile A，高端平台）和一加 3（标记为 Mobile B，中端平台）都采用了 ARM 的大小核架构。其 CPU 由两个簇组成：一组是性能较弱但是相对更省电的小核心（LITTLE），另一组是性能强但是更耗电的大核心（big）。后文中用 big.LITTLE 来指代这种大小核心的架构。所有的 big.LITTLE 的核心都共享内存，且工作负载可以在运行时无感的被操作系统在大小核心之间调度。注意 Mobile A 使用了高通第一代基于 Arm Cortex-A 系列定制的 64 位四核 Kryo CPU[168]。Redmi Note 4X （标记为 Mobile C，低端平台）由八个同样的小核心 CPU 核组成。表 3-2描述了三个平台详细的配置。在实验中，所有的后台应用被清空，屏幕关闭，手机切换到飞行模式来避免其他进程的资源竞争。

实验中也考虑了其他类型的移动设备，本文选择了 Jetson TX2[172] 作为边缘设备智能计算平台的代表。它有一个 256CUDA 核心的 NVIDIA Pascal GPU，典型的功耗是 7.5 瓦。更多的关于 Jetson TX2 的参数可以参看网页[172]。还有一个搭载了麒麟 970 芯片的华为手机，其搭载了一个专用智能加速器（Neural Processing Unit，NPU）。

深度学习框架： 本实验使用了三个 DNN 框架来做实验验证，包括了 TensorFlow[38], MACE[41] 和 TFLite[43]。所有的三个 DNN 框架都使用 Android NDK r14b 进行了编译，且编译的目标二进制架构为 *arm64-v8a*。TensorFlow（版本为 r1.7）用来衡量移动端和云端 CPU 的性能，其后端的 DNN 库为 Eigen3[173]。由于 TensorFlow 不支持移动端 GPU，本实验使用 MACE[41] 和 TFLite[43] 来衡量移动端 GPU 的性能。这两个 DNN 框架是通过厂商提供的 OpenCL 库实现对移动端 GPU 的调用的。

3.4　影响推理性能的因素

在本实验分析了五种典型的影响 DNN 模型在移动平台推理性能的因素。

3.4.1　因素 1: 计算单元

通过在配置界面中设置处理单元，DNNTune 可以将 DNN 模型运行到 CPU 或者 GPU 上。

3.4.1.1　延迟

图 3-5描述了 DNN 模型在三个移动平台的 CPU 和 GPU 上端到端的推理延迟，其中蓝色柱子代表高端的 Mobile A，橘色柱子代表中端的 Mobile B，灰色柱子代表低端平台 Mobile C。本文使用 TensorFlow 来衡量 CPU 的性能，使用 MACE 来衡量 GPU 的性能。

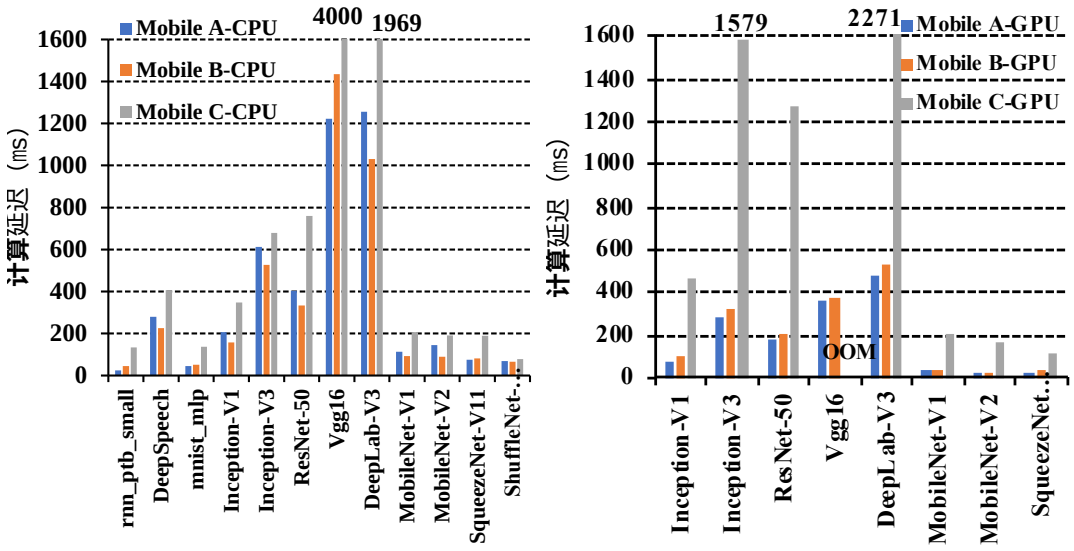


图 3-5 DNN 模型在 CPU 和 GPU 上最优配置下的延迟

Figure 3-5

Computation latency of DNN models on mobile CPUs and GPUs with optimal configuration

从图 3-5中可以得到如下的观察：**观察 1：现代的移动端 CPU 的计算能力足够处理小的 LSTM 或 MLP 模型，但是不能够有效的处理大的 LSTM 或 MLP 模型，且只能处理专门为移动端优化过的 CNN 模型。**

如图 3-5所示，小 LSTM 模型（rnn_ptb_small）和 MLP（minist_mlp）这两个模型在三个移动端 CPU 上的延迟变化范围是 24 毫秒到 139 毫秒，延迟较低因而可以直接部署在移动端 CPU 上。通常认为延迟小于 200 毫秒可以直接部署在手机上。而对于大的 LSTM 或 MLP 模型，其计算延迟从 215 毫秒到 400 毫秒，在移动端 CPU 上不能达到实时的处理。而对于大的 Transformer 模型，三个移动平台上计算的延迟为 6903 毫秒到 15821 毫秒，完全不能做到实时响应。注意，Transformer 模型延迟超过了图中延迟的最大值，因而没有被完整的画出。此外，传统的 CNN 模型（Inception-V1, Inception-V3, Vgg16, ResNet-50 and DeepLab-V3），其延迟的范围从 157 毫秒到 4000 毫秒不等，也同样因为模型计算规模太大而难以部署到移动端 CPU 上。

LSTM 和 MLP 模型的计算量和隐藏层单元数量、时间步以及模型的层数有关。以 LSTM 模型为例，小的 LSTM 模型 rnn_ptb_small 有 20 个时间步和 200 个隐藏单元，而大的 LSTM 模型 DeepSpeech，有 16 个时间步和 4096 个隐藏单元。因此，一个 LSTM 模型能否被直接部署到移动端 CPU 取决于其模型的架构和参数。

传统的 CNN 模型通常是设计用来达到最高图片分类的准确率，计算量很大，通常部署在云端。面向移动端优化的 CNN 模型，例如 MobileNet-V1、MobileNet-

V2、SqueezeNet-V11 和 ShuffleNet-V2 0.5×，通常是设计用来在资源受限的移动平台和嵌入式设备上运行，同时又保证一定的准确率。

在移动设备上，GPU 也被集成进 SoC 中，通常用于驱动显示屏，最近的研究者们开始将计算密集型的 DNN 模型迁移到移动端 GPU 上运行[108]。图 3-5b中展示了 DNN 模型在三个移动端 GPU 上运行的实验结果，其运行的 DNN 框架是 MACE。由于 MACE 不支持 LSTM 和 MLP，这两类模型的结果没有展示出来。

从图 3-5b中可以得到如下的观察：**观察 2：高端和终端的移动端 GPU 与 CPU 相比可以显著的推理延迟，但是低端平台不能。**

如图 3-5b所示，MobileNet-V1 模型在 Mobile A-GPU 和 Mobile B-GPU 上的延迟分别是 32 毫秒和 40 毫秒，与其 SoC 中集成的 CPU 相比，使用 GPU 可以分别获得 3.5× 和 2.3× 的加速比。所有的模型加速比求平均，两个移动平台上 GPU 可以获得 3.3× 和 2.3× 的加速比。原因是因为高端和中端的 GPU 有 256 个计算单元，而对应的 CPU 分别只有 12 个和 6 个计算单元, 其中每个小核心有 1 个算术逻辑单元（Arithmetic Logic Unit, ALU），每个大核心有 2 个 ALU。

然而, 在低端的移动 GPU 上情况非常不同。注意, 当在 Mobile C-GPU 上运行 Vgg16 模型的时候出现了内存不足的错误（Out of memory error，OOM），Mobile C 只有 3GB 内存，而 Mobile A 和 Mobile B 分别有 8GB 和 6GB 内存。在 Mobile C 平台上，GPU 相比 CPU 的平均加速比只有 0.9×，这意味着低端的 GPU 甚至比 CPU 更慢。这是因为低端平台的 GPU 只有 96 个计算单元。此外，OpenCL-Z[174] 的基准测试结果也表明，Mobile C-GPU 的峰值性能只有 32.55GFLOPS，而作为对比，Mobile A-GPU 的峰值性能是 256GFLOPS。

Facebook 也有类似的的发现，即在中端的计算平台上，GPU 也只能提供和 CPU 相近的理论峰值性能。Mobile C 是一个低端平台的设备，其 GPU 计算的延迟甚至会高于 CPU。对于 DeepLab-V3，Mobile C-CPU 的性能要高于 Mobile C-GPU，这是因为 CPU 上的 DNN 算子的 kernel 是经过高度优化的（有成熟的库），而 GPU 上 OpenCL 的库没有经过特殊的优化。

3.4.1.2　功耗

图 3-6描述了所有的 8 个模型在三个移动平台的 CPU 和 GPU 上的推理一次的能耗。其横轴表示 DNN 模型，纵轴表示具体的能耗值（单位：焦耳）。从图 3-6中可以得到如下观察：**观察 3：移动端 GPU 比 CPU 的性能功耗比要更高，特别是对于高端和终端的 GPU。**

如图 3-6所示，以 MobileNet-V1 为例，其在三个平台 CPU 上运行的能耗分别是 0.22、0.51 和 0.35 焦耳, 而在三个平台 GPU 上运行的能耗分别只有 0.062、0.063 和 0.117 焦耳。所有 DNN 模型的能耗数据平均来看，在三个移动平台 Mobile A、Mobile B 和 Mobile C 上 CPU 消耗的能量分别是 GPU 的 2.9×、7.5× 和 2.1×。

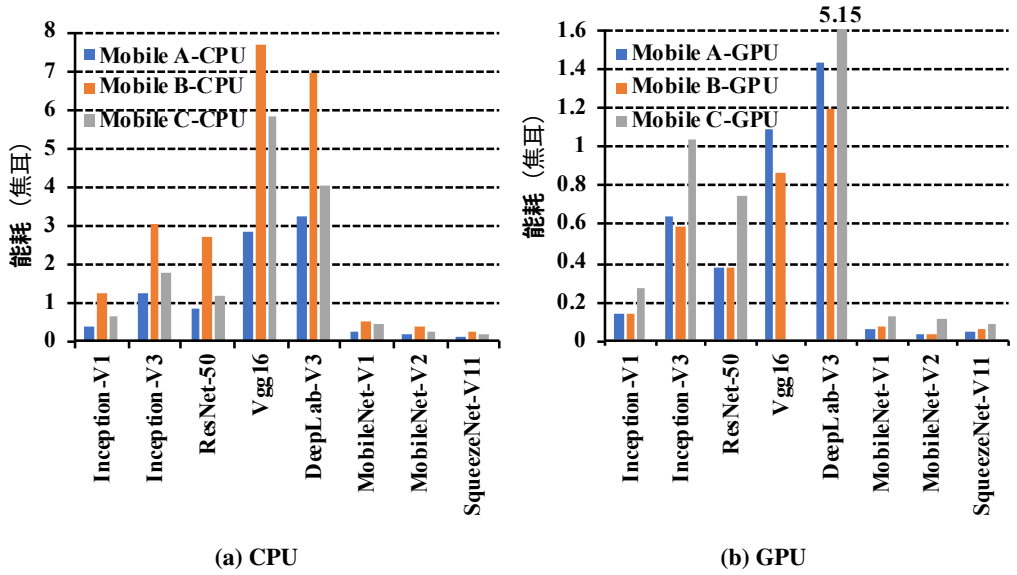


图 3-6 三个移动平台上的能耗

Figure 3-6 Energy consumption of models on three mobiles

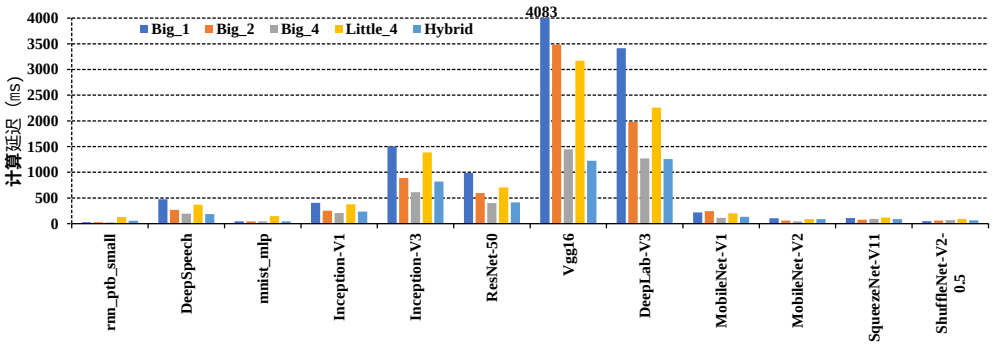


图 3-7 Mobile A-CPU 上 DNN 推理延迟与 CPU 大小核心及线程数变化

Figure 3-7 Performance on Mobile A-CPU varying with CPU affinity and thread number

3.4.2 因素 2：CPU 大小核心

本实验将基于 TensorFlow 和 Eigen 库分别衡量大核心、小核心和“大核心 + 小核心”（为了简便，后文用 big.LITTLE 代替）三种配置。

3.4.2.1 延迟

图 3-7展示了在 Mobile A-CPU 上的计算延迟。其中分别展示了（1）当只使用大核心时推理延迟的变化；（2）当使用不同的 CPU 核心（大核心、小核心、big.LITTLE）三种配置时的推理延迟。

从图 3-7中可以得到如下的观察：**观察 4：同时使用大核心和小核心不一定会降低推理延迟。**如图 3-7所示，对于 Inception-V3 模型，big.LITTLE 的执行延迟比只使用大核心要更慢。DeepLab-V3 中也类似。这是因为当同时使用大小核心的时候，工作负载会被 Eigen 库自动的平均地分配给 CPU 核心来执行，而库本身并不能感知到底层的四个大核心和四个小核心性能的差异。DNNTune 观察

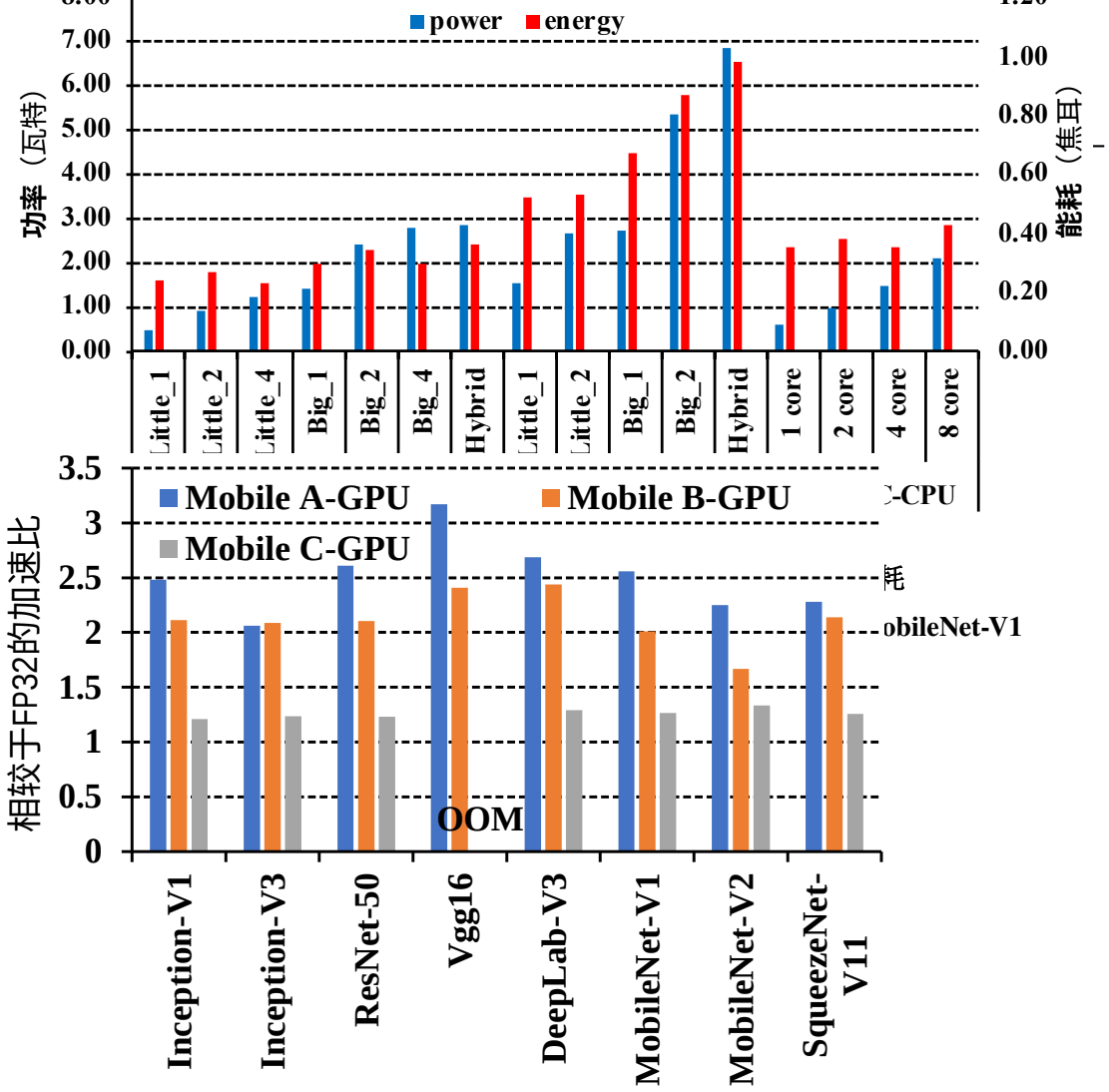


图 3-9 FP16 相比 FP32 的加速比

Figure 3-9 Speedup of FP16 over FP32

到当只使用大核心时，其 CPU 利用率可以达到 75% 以上；而在 big.LITTLE 场景下大核心 CPU 利用率降低到了 36%。

3.4.2.2 能耗

图 3-8展示了 MobileNet-V1 在三个平台 CPU 上的能耗。其中蓝色的柱子代表功率（左边的坐标轴），红色的柱子代表能耗（右边的坐标轴）。

对于 big.LITTLE 架构，大核心的功率要远远高于小核心。例如，Mobile A 上一个小核心的功率是 0.5 瓦特，而一个大核心的功率是 1.41 瓦特。本实验发现一个小核心相比一个大核心可以节省 18.3% 的能耗，使用公式 3-1可以揭示这个原因。尽管一个小核心的功率比一个大核心的功率要低很多（0.50 瓦特对比 1.41 瓦特），但是他的延迟也更高（483 毫秒 209 毫秒）。因此，总体的能耗对比：小核心为 $0.50 \times 483 \rightarrow 241.5$，而大核心为 $1.41 \times 209 \rightarrow 294.69$。注意到高端平台 Mobile A 单核的性能比中端平台 Mobile B 的功率更低，这是因为 Mobile A 的 SoC 是基于 10 纳米 FinFET 工艺制造的，而 Mobile B 是基于 14 纳米工艺制造的。此外 Mobile A-CPU 的微架构相比 Mobile B-CPU 可以节省 30% 的能耗[175]。

3.4.3 因素 3：CPU 线程数量

本节衡量 DNN 模型在线程上的可拓展性。

3.4.3.1 延迟

图 3-7表明，当只使用大核心的时候，CNN 的模型展示了良好的随着线程数增多的可拓展性。其加速比与使用的线程数基本成正比。然而，rnn_ptb_small 和 mnist_mlp 这两个模型没有加速，这是因为这两个模型主要计算是矩阵向量乘且规模相对较小，而 Eigen 的库没有专门针对小规模矩阵向量乘来做优化以充分利用多核的计算能力。小核心上也表现出了类似的情况。

3.4.3.2 功耗

基于公式 3-1，图 3-8中展示了三个设备上的能耗。**观察 5：使用更多的核心不总是消耗更多的能量。**原因是因为能耗是系统功率和执行时间的乘积。例如，在 Mobile A-CPU 上，与使用一个小核心相比，使用四个小核心将计算延迟从 483 毫秒降低到了 183 毫秒。尽管使用系统功率从一个核心的 0.5 瓦特增加到了四个核心的 1.41 瓦特，总的能耗也只降低了 8%。DNNTune 能够自动的根据用户的目标（延迟最低或功耗最低）来自动的调整运行的配置。

表 3-3 七个 CNN 模型在四个不同的 DNN 框架上的延迟、功耗和内存占用

Table 3-3 Computation latency, energy consumption and memory usage of 7 CNN models with 4 different DNN frameworks

	Inception-V1			Inception-V3			ResNet-50			Vgg16			DeepLab-V3			MobileNet-V1		
	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M	Lat	En	M
TF	207	0.413	110	612	1.812	310	403	1.126	340	1446	5.881	1200	1256	5.259	150	113	0.296	70
TFLite	216	0.663	50	865	2.876	120	479	1.805	120	1959	8.365	630	1547	5.047	80	124	0.204	30
MACE	87	0.459	90	574	2.958	100	196	1.023	150	432	2.169	400	843	4.276	100	45	0.237	20
Quant	50	0.194	10	216	0.963	30	122	0.570	30	446	2.202	160	720	3.146	20	24	0.102	10

3.4.4 因素 4：DNN 框架

本节展示了如何使用 DNNTune 来衡量模型在三个不同框架（TensorFlow、TFLite、MACE）上的端到端的延迟（Latency，标记为 Lat，单位：毫秒）、功耗（Energy，标记为 En，单位：焦耳）和内存占用（Memory occupancy，标记为 M，单位：MB）。表 3-3展示了在 Mobile A-CPU 上使用四个大核心，在三个 DNN 框架上的运行结果。其中，TFLite 同时支持浮点数（float32，简写为 FP32）的模型和量化（8 位整数，简写为 INT8）的模型。

3.4.4.1 延迟

对于 FP32 的 DNN 模型，MACE 是能达到最低延迟的框架。这是因为 MACE 框架对各种常见形状的卷积操作都使用 SIMD 指令做了特殊的优化，而 TFLite

则采用“img2col+gemm”（先将特征向量转成一个矩阵，然后使用通用矩阵乘计算卷积结果）的方式计算所有的卷积操作。因而，没有专门针对 DNN 模型中的卷积操作进行特殊的优化。然而，当考虑使用量化模型时，TFLite 的 INT8 量化模型能够达到最低的计算延迟。

3.4.4.2　功耗

当考虑功耗的时候，没有一个框架能够在所有模型上都达到最低的功耗。以两个 DNN 模型为例：对于 MobileNet-V1，TFLite 是最优的选择，每次推理只消耗 0.203 焦耳的能耗，而对于 ResNet-50，MACE 是最优的。

3.4.5　内存占用

与 TensorFlow 相比，MACE 和 TFLite 占用的内存都更低，它们是面向资源受限的移动平台专门优化的，因而会进行仔细的内存管理以尽可能减少内存占用。与 FP32 的模型相比，INT8 的量化模型能够平均节省 74% 的内存占用。这是因为量化模型权值的一个元素使用 8 比特，而浮点数模型使用 32 个比特。

3.4.6　因素 5：GPU 半精度

GPU 通常支持半精度浮点数（float16，简写为 FP16）来达到更高的峰值性能，而精度损失通常可以忽略[32]。在本节使用 FP16 来衡量 DNN 模型推理的性能。在 Mobile C-GPU 上，由于 Vgg16 模型出现了内存不足的错误，因而 Vgg16 的数据是缺失的。对于 Inception-V1 模型，使用 FP16 在 Mobile A-GPU 和 Mobile B-GPU 分别能够达到 FP32 的 2.5× 和 2.2× 倍。性能的提升来自于两个方面。第一，高端的 GPU 上有对 FP16 的硬件支持[176]；第二，FP16 减小了权值的大小，因此降低了对内存的访问。使用 Snapdragon_Profiler，本实验发现 Mobile A-GPU 上的最大内存带宽从 2.9GB/s 降低到了 2.3GB/s。因此，对于高端和中端的移动平台，使用 FP16 精度的 DNN 模型既能够减少延迟又能够减少内存占用。然而，低端的平台使用 FP16 加速效果没有高端和中端平台那么明显（只有 1.25× 的加速比），这是因为低端的 Adreno GPU 不能原生支持 FP16。

3.4.7　讨论：衡量智能处理器

本节讨论两个典型的硬件平台来衡量 DNNTune 对智能处理器的支持。一个是 NVIDIA Jetson TX2[172]，其包含一个 256 核心的 Pascal 架构的 GPU；另一个是华为荣耀 10[177]，其华为麒麟 970 的 SoC 中包含了一个神经处理单元（Neural Processing Unit，NPU）。为了在 NPU 上运行，本实验使用 HiAI SDK[178] 将 Caffe 的模型（.caffemodel）和 TensorFlow 的模型（.pb）转为特定的格式（.cambricon）。

表 3-4展示了在 5 个 CNN 模型上，智能处理器的延迟和加速比。本实验以 Mobile A 的 CPU 和 GPU 为基准计算智能处理器的加速比。具体而言，最优的 CPU 配置是在 MACE 平台使用四个大核心和 MACE 作为 DNN 框架。表 3-4表明，除了 MobileNet-V1 和 SqueezeNet-V11，对于大多数的 CNN 模型，智能

表 3-4 DNN 在 Jetson TX2 和华为 NPU 上的性能

Table 3-4 DNN performance on Jetson TX2 and NPU

计算平台	DNN 模型	Inception-V1	Inception-V3	ResNet-50	MobileNet-V1	SqueezeNet-V11
NPU	计算延迟（毫秒）	35	71	53	33	13
	相比最优移动 CPU 的加速比	2.49×	8.08×	3.70×	1.76×	0.79×
	相比最优移动 GPU 的加速比	2.27×	3.91×	3.25×	0.96×	1.81×
Jetson TX2	计算延迟（毫秒）	17	91	59	15	12
	相比最优移动 CPU 的加速比	5.12×	6.3×	3.70×	3.83×	0.86×
	相比最优移动 GPU 的加速比	4.67×	3.05×	3.32×	2.11×	1.96×

处理器能够达到显著的性能提升。对于 Inception-V1 和 ResNet-50，NPU 相比 CPU 能够分别达到 2.49× 和 8.08× 的加速比，相比 GPU 能够达到 2.27× 和 3.91× 的加速比。Jetson TX2 上可以看到相似的结果。智能处理器对 MobileNet-V1 和 SqueezeNet-V11 的加速并不明显，这是因为这两个模型中更多的使用了深度可分离卷积（Depthwise Convolution），其计算量相比传统的卷积低一个数量级，因而不能充分发挥智能处理器的计算能力。

3.5 端云协同的自动调优

本实验分别使用了三个移动平台 Mobile A、Mobile B、Mobile C 的 CPU 和 GPU，以及一个服务器上的 CPU 作为云端平台，其 CPU 是 Intel Xeon E5-2620 v4。本实验使用 TensorFlow 来衡量 CPU，使用 MACE 来衡量移动端的 GPU。在云端平台，MobileNet-V2、SqueezeNet-V11、Vgg16 和 ShuffleNet-V2-0.5× 的计算延迟分别是 38 毫秒、10.7 毫秒、90 毫秒和 5 毫秒。对于端云之间的通信，分别考虑 3G、4G 和 Wi-Fi 条件下的数据传输。这三种无线条件下，传输上述三个网络输入数据的网络延迟分别是 850 毫秒、180 毫秒和 100 毫秒。

3.5.1 延迟优先

图 3-10分别展示了端云协同的三种方式。对于端云协同的切分，端到端的延迟划分为三个部分：DNN 模型前半部分在移动端设备上的执行延迟，一层的输入数据和结果传输的延迟，以及在 DNN 模型后半部分在云端上的延迟。DNNTune 遍历 DNN 模型所有的层，来决定最优的划分策略。

端云协同处理最优。如图 3-10(a) 中，对于 MobileNet-V2，*bottleneck_3_3* 是最优的划分点，端云协同端到端的延迟是 115 毫秒。作为对比，只在移动端运行和只在云端执行的延迟分别是 154 毫秒和 138 毫秒。图 3-11展示了三个网络每层的输入数据大小和计算时间。可以看到 *bottleneck_3_3* 的输入数据比较小，因此传输到云端的开销会比较小，其延迟为 20.3 毫秒。作为对比，把原始输入数据传输到云端的开销是 100.7 毫秒。与此同时，在切分点之后还有很多计算开销（104.1 毫秒在移动端），因而 DNN 切分点之后的部分在云端执行相比移动端就

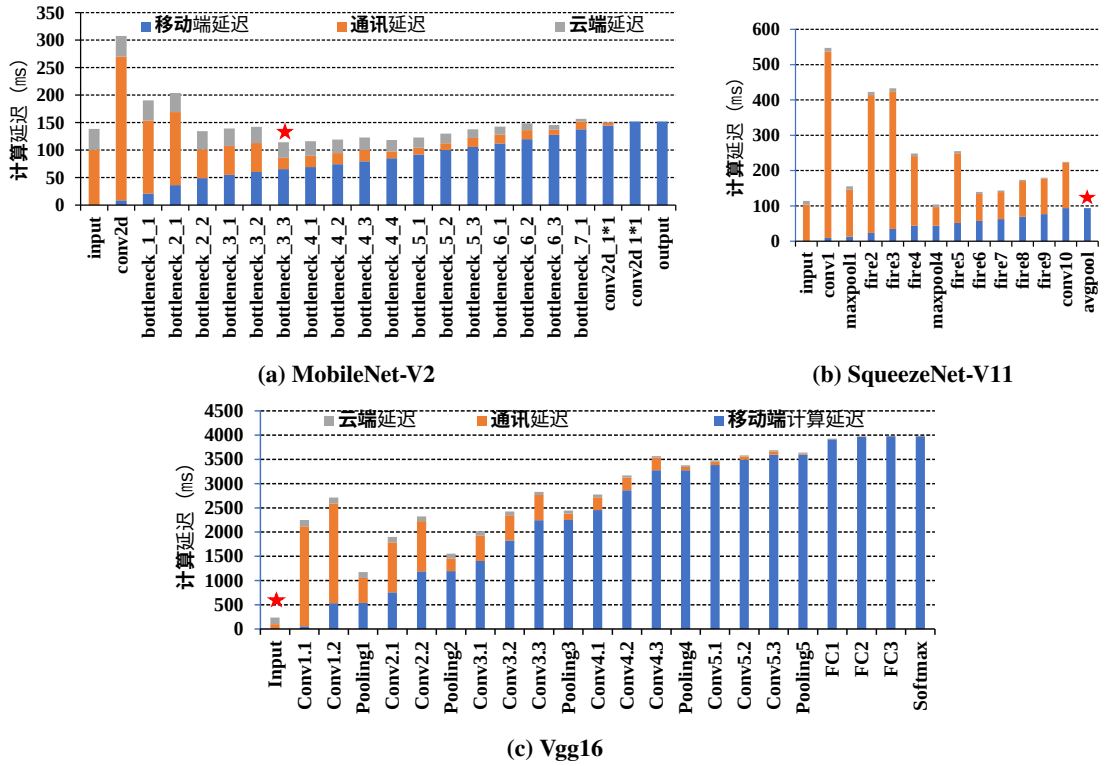


图 3-10 在 mobile CPU-A 上基于 WiFi 环境下的端云协同的执行

Figure 3-10 Mobile-cloud partitioning latency on mobile A-CPU via Wi-Fi

能获得极大的性能收益（在云端执行后半部分的延迟是 33.2 毫秒）。因此，在 *bottleneck_3_3* 这一层切开能够获得比较大的性能提升。

只在移动端执行最优。如图 3-10(b) 所示，对于 SqueezeNet-V11，只在移动端执行是最优的。原因是 SqueezeNet-V11 在移动端的时间非常短（99.7 毫秒），而迁移到云端执行的端到端延迟是 110.7 毫秒。

只在云端执行最优。如图 3-10(c) 所示，对于 Vgg16，只在云端执行是最优的。有两个原因，第一，如图 3-11(c) 所示其输入数据的数据量比较小；因而其传输到云端的数据传输开销相对于计算的开销较小。第二，其总体的计算量比较大，所以在云端执行比在移动端执行能够获得更大的加速比。

更多平台的实验结果。

除了 MobileNet-V2，另一个端云协同执行的模型是 ShuffleNet-V2，其最优的切分点是 *MaxPool_2*，端到端的延迟是 86 毫秒，而其只在移动端和只在云端处理的延迟是 112 毫秒和 105 毫秒。

在 Mobile-B 和 Mobile-C 上也能获得相似的结果。如表 3-5所示，Mobile A-CPU 能够达到 1.2× 的加速比。而 ShuffleNet-V2 0.5× 在 Mobile B-CPU 上，只在移动端执行能够达到最低的延迟，因为计算的延迟比通信延迟更低。对于 ResNet-50、Inception-V3、DeepSpeech 和 DeepLab-V3，这些网络在移动端执行的延迟很高，在云端执行是最优的部署策略。而对于 SqueezeNet-V11 这样的小模型，只

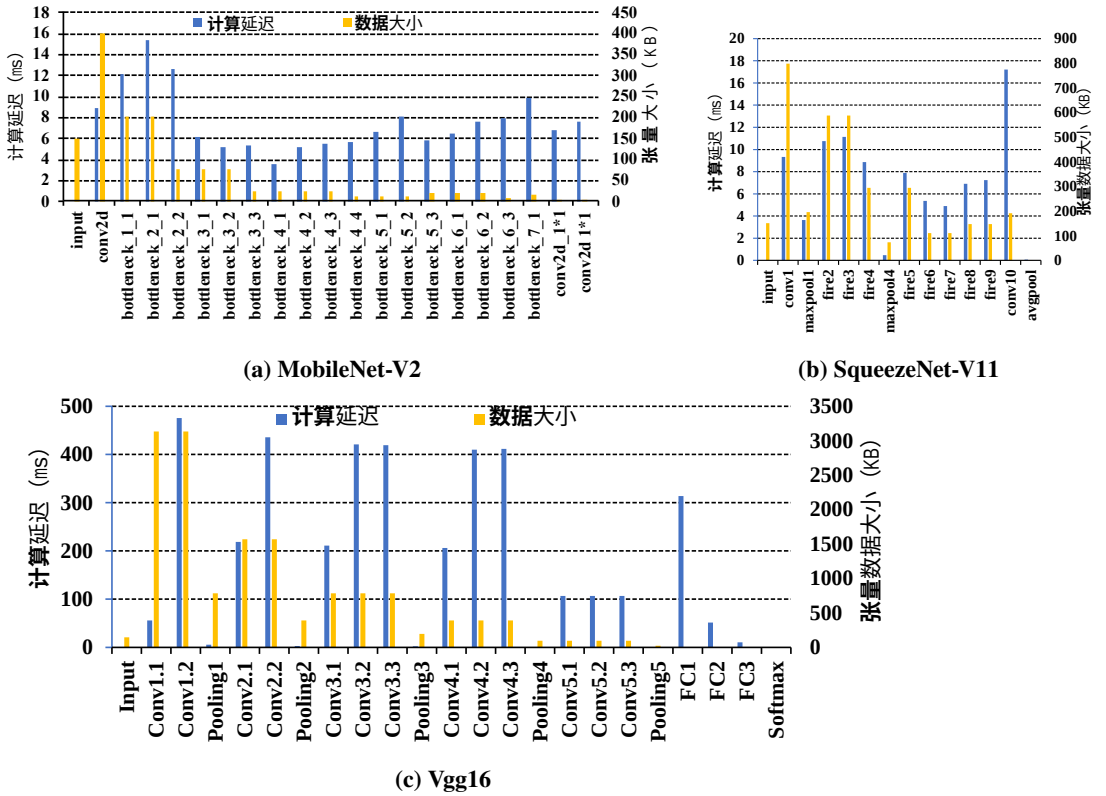


图 3-11 三个网络在 Mobile-A CPU 上按层的计算延迟和每层的输入数据的大小

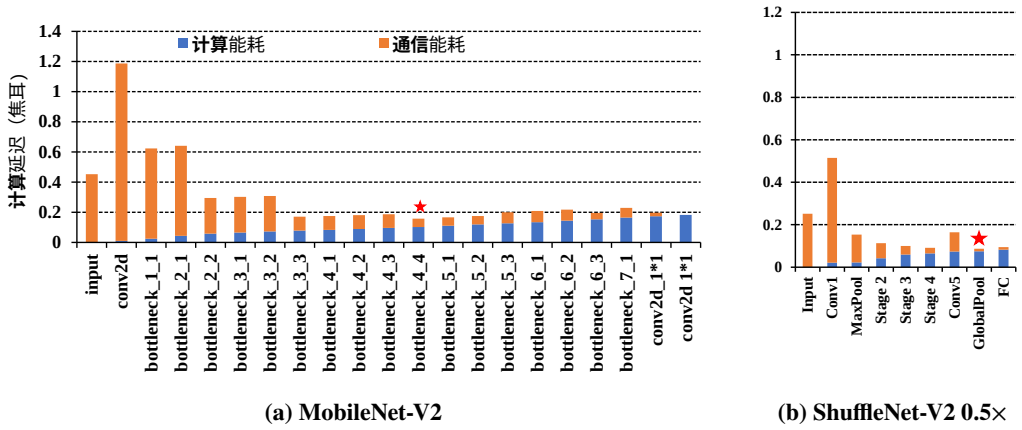
Figure 3-11 Layer-wise computation and layer input data size latency on Mobile A-CPU

在移动端执行是最优的。直观的讲，如果一个模型的前几层能够快速的减小层之间中间结果数据量的大小，而后面的层依然需要大量的计算，这样端云系统的执行只需要传输少量的数据，因而这个模型能够从端云协同的运行中受益。而对于 MLP、LSTM 和 Transformer 类模型，在云端执行是最优的。原因有两方面：一方面这些模型的输入数据与图像类相比数据量很小；第二这些模型在云端的推理延迟要远低于在移动端的推理延迟。

表 3-5 三个移动平台 Wi-Fi 条件下端云协同的切分

Table 3-5 Mobile-cloud partitioning on three mobile platforms using Wi-Fi.

		MobileNet-V2					ShuffleNet-V2 0.5×				
	配置	移动端	云端	端云协同	加速比	切分点	移动端	云端	端云协同	加速比	切分点
Mobile A-CPU	1 big	154	138	115	1.2×	bottleneck_3_3	42	105	86	1.0×	output
	1 little	381	138	138	1.0×	input	112	105	86	1.23×	MaxPool
Mobile B-CPU	1 big	136	138	125	1.10×	bottleneck_5_2	33	105	33	1.0×	output
	1 little	181	138	138	1.09×	input	65	105	65	1.0×	output
Mobile C-CPU	1 core	358	138	138	1.0×	input	99	105	81	1.22×	Stage2
	8 core	190	138	138	1.0×	input	66	105	66	1.0×	output
Mobile C-GPU	GPU	152	138	83	1.66×	bottleneck_4_4	100	105	64.6	1.55×	MaxPool



(a) MobileNet-V2　　(b) ShuffleNet-V2 0.5×

图 3-12 4G 下 mobile A-CPU 端云协同切分策略的能耗

Figure 3-12 mobile-cloud partitioning energy consumption on mobile A-CPU via 4G

3.5.2　功耗优先

本节将优化的目标改为功耗优先。使用如下的公式来计算能耗：

$$E = T_m(e) \times p_m(e) \tag{3-1}$$

其中 E 是执行整个 DNN 推理的能耗，单位为焦耳（Joule），e 是移动平台，m 是 DNN 模型，$p_m(e)$ 是在 e 上执行 m 时的系统平均功率，通过功耗收集器获得。首先记录一遍没有工作负载时平台的功率，记为 $p(e_{idle})$；然后，开始执行工作负载 m，并记录系统功率，记为 $p(e_{active})$；最后，用公式 $p_m(e) = p_m(e_{active}) - p(e_{idle})$ 来计算运行 m 时的功率。本节用 TensorFlow 来作为编程框架，使用三个移动端 CPU 来衡量实验结果。3G、4G 和 Wi-Fi 模组的功率分别是 0.8、2.5 和 1.2 瓦特。将原始的输入数据（图片）传输到云端的功耗分别是 0.712、0.450 和 0.121 焦耳。

3.5.2.1　4G

图 3-12展示了 Mobile A 在所有可能的切分点进行端云协同执行的功耗。当切分点从左往右移动的时候，因为更多的计算被划分到了移动设备上，移动设备计算消耗的能量增加。而数据传输的能耗与被传输的张量的大小正相关。实验结果表明，对于 MobileNet-V2，*bottleneck*_4_4 是最优的切分点，其能耗为 0.158 焦耳。而只在云端执行和只在移动端执行的功耗分别是 0.453 和 0.182 焦耳。对于 ShuffleNet-V2，*GlobalPool* 是最优的切分点，其能耗为 0.091 焦耳，只在云端执行和只在移动端执行的功耗分别是 0.252 和 0.099 焦耳。

3.5.2.2　Wi-Fi

对于 MobileNet-V2，当只使用 Wi-Fi 的时候，在云端执行是最优的策略。这是因为 Wi-Fi 的延迟很低，因而只消耗很低的功耗就能完成通信 (0.121 焦耳)，而只在三个移动端执行的功耗分别是 0.182、0.318 和 0.268 焦耳。对于 ShuffleNet-V2，如表 3-6所示，端云协同的执行可以节省 5.5% 到 15.1% 的能耗。

表 3-6 ShuffleNet-V2 0.5× 在三个移动 CPU 上在 Wi-Fi 下的能耗

Table 3-6 Energy consumption on three mobile CPUs using Wi-Fi for ShuffleNet-V2 0.5×

	只在移动端	只在云端	端云协同	节省的能耗	切分点
Mobile A-CPU	0.088	0.121	0.077	12.5%	Stage2
Mobile B-CPU	0.099	0.121	0.084	15.1%	Stage2
Mobile C-CPU	0.055	0.121	0.052	5.5%	Stage4

3.5.2.3 3G

在 3G 的网络下，将输入数据传输到云端的功耗是 0.712 焦耳，而 MobileNet-V2 在 Mobile A-CPU、Mobile C-CPU 和 Mobile C-CPU 上的执行功耗分别是 0.182、0.318 和 0.268 焦耳。因此，在 3G 条件下，只在移动端执行是最优的，因为传输输入到云端的功耗比在移动端上计算所消耗的能量要更多。

3.6 多计算单元异构并行的自动调优

移动平台上集成了越来越多的异构计算单元[179]，并由多种异构单元封装成一个片上系统（System on Chip, SoC）。例如，高通骁龙 710（Snapdragon 710）[180]移动平台包括 2 个 ARM 大核心，6 个 ARM 小核心，一个 Adreno 616 GPU 和一个加速器 Hexagon 685 的数字信号处理器（Digital Signal Processor, DSP）。这些移动平台在计算能力、功耗、内存和编程接口方面差异很大[19]。根据章节 3.4性能基准测试的结果，本文有如下的核心观察：**目前的移动端编程框架，不能自动的把 DNN 的模型的算子映射到多个异构计算单元并行执行，以达到最优的性能。**例如在章节 3.4.2中，由于上层的编程框架不能感知到 CPU 大小核心的性能差异，同时使用大核心和小核心不一定会降低推理延迟。

为了能够解决这些问题，Han 等提出了 MOSAIC[49]，使用异构敏感、通信敏感和约束敏感的模型切片的方法来把 DNN 的模型分配到不同的异构计算单元上。然而，MOSAIC 把 DNN 模型当做一连串的线性的模型，而不考虑挖掘算子之间的并行度。DNN 模型可以看做一个有向无环图（Directed Acyclic Graph, DAG），其算子是节点、算子之间的数据（张量）依赖为边。把算子映射到一系列计算设备上可以看做是一个 DAG 图的调度问题。在传统的 CPU+GPU 的异构服务器上，研究人员提出了许多的动态调度并行任务到异构硬件上的方法和策略，包括[181–185]。其中代表性的工作是 StarPU[186]。StarPU 实现了一个基于列表调度（list scheduling）或者工作偷取（work stealing）策略的通用的静态异构调度框架。然而，这些运行时的启发式运行时并不适用于移动的平台。因为 DNN 模型有一些特殊的拓扑结构且通常是静态的，通常是由一系列的子图（或者块结构）来构成整个大的模型。可以基于这些结构特征来获取接近最优的解决方案。但是 StarPU 不能感知到 DNN 模型 DAG 图的拓扑结构特征，因而失去了获取更好的

调度策略的机会，不能够更进一步的降低推理延迟。

本节提出了多异构计算单元感知的自动调优（为了和端云协同的调优区分开，命名为 HOPE），把 DNN 模型的算子映射到不同的计算单元上，在不同的异构计算单元上并行的执行多个算子。本文的核心观察是许多 DNN 模型有算子间的并行，因此没有数据依赖之间的算子可以在不同的计算设备上并行执行。同时，DNN 模型的拓扑结构可以帮助寻找最优的调度策略。具体而言，HOPE 首先测量模型中每个算子的计算延迟和张量在不同异构计算单元之间传输的开销；将图调度的问题抽象为了一个整数线性规划（Integer Linear Programming，ILP）求解的问题，并且提出了一个启发式算法对 DNN 的计算图进行预处理。调度器能够把 DNN 模型中的算子映射到不同的计算单元上。之后，本文设计实现了一个执行引擎，可以不断的把算子的计算代码运行到对应的计算单元上。HOPE 基于 MNN[42] 实现了 HOPE 的运行时部分，其调度器基于 Python 实现，与运行时分离，因而可以被集成到其他的移动端 DNN 推理框架。

3.6.1　问题形式化

在多个异构计算设备上调度执行数据流图的问题可以被形式化为如下的问题：给定一个数据流图 $G(V, E)$, 和包含目标平台所有计算设备的集合 D, 对于每个节点 $v_i \in V$，其在设备 d_j 上的执行时间设为 $cl_{i,j}$; 对于一对异构计算设备 (d_i, d_j)（$d_i, d_j \in D$），将张量 T 从设备 d_i 传输到 d_j 的传输开销设为 $C_(d_i, d_j)(T)$。HOPE 的目标是求出一个执行计划 $\mathfrak{R}$，将每个节点 v_i 映射到计算设备 d_j 上，以最小化端到端的执行延迟 $\tau(\mathfrak{R}, G)$。具体而言，一个执行计划 $\mathfrak{R}$ 可以被表示为 $\mathfrak{R}(b, \psi)$，其包含两个部分：第一部分是节点到计算设备的映射矩阵 b，第二个部分是一个表明节点之间在每个设备上的执行顺序的矩阵 ψ。在 $\mathfrak{R}(b, \psi)$ 中，b 是一个二值映射矩阵，记录了每个节点被映射到的计算设备。每个元素 $b_{i,j}$ 表示节点 v_i 是否被映射（调度）到设备 d_j：

$$b_{i,j} = \begin{cases} 1 & \text{if } v_i \text{ is dispatched on } d_j \\ 0 & \text{if } v_i \text{ is not dispatched on } d_j \end{cases}$$

而 ψ 是一个数据矩阵，其第 j 个元素是一个矩阵，表示在设备 d_j 上所有的算子之间的执行顺序，$\psi_{i,k,j}$ 如果为 1 则表示在设备 d_j 上节点 v_i 在节点 v_k 之前执行。

3.6.2　基于整数线性规划的理论最优解

在这一节中，首先将异构并行调度数据流图（抽象为有向无环图，Directed Acyclic Graph, DAG）的问题先形式化为整数线性规划（Integer Linear Programming, ILP）的问题。注意，在形式化过程中需要考虑在不同的设备之间通信开销的问题。之后，提出来了一个图划分的算法，来减少 ILP 求解器的时间复杂度。

3.6.2.1 ILP 表达式

节点执行延迟：一个节点 v_i 在一个设备 d_j 上的执行延迟由两部分组成。第一部分是节点的计算延迟 $cl_{i,j}$。如果计算设备不支持执行节点 v_i 的计算，则将设置 $cl_{i,j} = +\infty$。第二部分是从节点 v_i 的前驱接收张量的通信延迟。从节点 v_i 的前驱 v_j 接收张量的通信延迟标记为 $C_{(d(v_j),d(v_i))}(T)$，其中 $d(v_j)$ 和 $d(v_i)$ 分别代表 v_j 和 v_i 所被映射到的计算设备。T 代表从 $d(v_j)$ 传输到 $d(v_i)$ 的张量。更进一步的，如果 v_i 有多个前驱，那设置其所有的通信开销为每个前驱的通信开销之和，如下面的公式所示：$comm_i = \sum_{v_j \in pred(v_i)} C_{d(v_j),d(v_i)}(T)$。综合以上两部分的延迟，节点 v_i 在设备 d_j 上的执行延迟 $t_{i,j}$ 可以被形式化为如下的表达式：

$$t_{i,j} > b_{i,j} * cl_{i,j} + \sum_{(v_k,v_i)}^{\forall v_k \in pred(v_i)} (b_{i,j} - b_{k,j}) * C_{(d(v_k),d(v_i))}(T) \tag{3-1}$$

表达式 $(b_{i,j} - b_{k,j}) * C_{(d(v_k),d(v_i))}(T)$ 描述了前驱节点 v_i 和节点 v_k 之间是否有通信开销。如果节点 v_i 和节点 v_k 被映射到了不同的计算设备上（比如说 v_k 被映射到了 CPU 上 v_i 被映射到了 GPU 上），那么由 v_k 产生的张量需要被从 CPU 的内存空间传输到 GPU 的内存空间。注意，这个通信可以和与 v_i 没有依赖关系的算子有重叠（也就是并行执行）。

可以验证约束 3-1的正确性。

（1）如果 $b_{i,j} = 0$，约束 3-1可化简为

$$t_{i,j} > \sum_{(v_i,v_k)}^{\forall v_k \in pred(v_i)} (0 - b_{k,j}) * C_{(d(v_k),d(v_i))}(T)$$

，由于 $b_{k,j}$ 要么为 0 要么为 1，并且执行延迟 $t_{i,j}$ 是一个正数，所以表达式的右半部分小于等于 0，因此这个约束将总是为真；

（1）如果 $b_{i,j} = 1$，约束 3-1可化简为

$$t_{i,j} > cl_{i,j} + \sum_{(v_i,v_k)}^{\forall v_k \in pred(v_i)} (1 - b_{k,j}) * C_{(d(v_k),d(v_i))}(T)$$

不等式的右边正好就是 v_i 在 d_j 上的执行延迟。

目标函数：HOPE 的目标是最小化计算图的执行延迟，因此引入了一个辅助变量 $st_{i,j}$ 来描述节点 v_i 在时刻 $st_{i,j}$ 开始在设备 d_j 上执行。因此，模型端到端的执行时间就是从第一个节点的开始执行时间到最后一个节点执行完成的时间。HOPE 的目标是优化如下的端到端的延迟：

$$Minimize \quad \tau(\mathfrak{R}, G) \tag{3-2}$$

$$st.$$

$$\tau(\mathfrak{R}, G) > b_{i,j} * (st_{i,j} + t_{i,j}) \quad (\forall v_i \in V, \quad \forall d_j \in D)$$

节点的数据依赖关系约束：对于节点 v_i 和节点 v_k，如果有一条边 $(v_i, v_k) \in E$，那么如下的图中算子的数据依赖关系约束（反映到计算图上就是图的拓扑结构约束）必须要被满足：

$$st_{k,l} > st_{i,j} + t_{i,j} + (b_{i,j} - 1) * M \quad (\forall (v_i, v_k) \in E, \quad \forall d_l, d_j \in D) \tag{3-3}$$

约束 3-3描述了对于任意的节点 v_k 和任意的设备 d_j 上，只有 v_k 所有的前驱节点都被执行完了，v_k 才能开始执行。可以验证如果 $b_{i,j}$ 为 0，这条约束会为真。

设备执行节点的约束：受限于性能，移动平台上的计算设备不会同时运行多个算子，因此在任意时刻至多会有一个算子在一个设备上运行。并且算子的执行是非抢占式的。在 DNN 模型中，如果从节点 v_i 到节点 v_j 有一条路径，那么设备执行节点的约束会自然被满足，因为图中节点执行先后顺序的约束会自然的保证 v_j 会在 v_i 之后完成。因此，只需要在互相之间没有依赖关系的节点之间引入约束：

$$st_{i,j} > st_{k,j} + t_{k,j} \quad \forall d_j \in D \quad or \quad st_{k,j} > st_{i,j} + t_{i,j} \quad \forall d_j \in D \tag{3-4}$$

这个约束表明了 v_i 和 v_k 可以以任意顺序执行，但是不能在同一个设备 d_j 上并行执行。换句话说，在同一个设备 d_j 上 v_i 和 v_k 是互斥的。引入一个辅助变量 $\psi_{i,k,j}$ 和一个足够大的正数 M 来改写上面的约束，以消除“或（or）”这样的自然语言描述。如下所示：

$$\begin{aligned} &st_{i,j} > st_{k,j} + t_{k,j} - \psi_{i,k,j} * M \quad \forall d_j \in D \\ &st_{k,j} > st_{i,j} + t_{i,j} - (1 - \psi_{i,k,j}) * M \quad \forall d_j \in D \end{aligned} \tag{3-5}$$

其中 $\psi_{i,k,j}$ 是一个二值变量，描述了 v_i 和 v_k 的执行顺序。如果 $\psi_{i,k,j}$ 为 0，v_i 会在 v_k 之后被执行，否则的话相反。

节点执行的约束：对于任意给定的模型，每个节点只需要执行一次，因此有下列的约束：

$$\sum_{j}^{\forall d_j \in D} b_{i,j} = 1, \quad \forall v_i \in V \tag{3-6}$$

总结：下图展示了 ILP 的表达式，基于约束 3-1到约束3-6的表达式，可以使用一个标准的 ILP 求解器，例如 GLPK[187] 来求解出结果。最后，执行计划是用两组变量来编码的：$b_{i,j}$ 描述节点映射到的设备，$\psi_{i,k,j}$ 描述了 v_i 和 v_k 之间的执行顺序。

3.6.3　启发式图划分

整数线性规划算法能够得到理论上的最优解。但是随着 DNN 模型中节点和边数量的增多，ILP 中的约束和变量也会随之急剧的增多，使用 ILP 算法求解问题的复杂度也会呈指数级增加，求出问题的时间也会急剧增加。例如，Inception-V4[188] 模型中有 7 个节点的子图，ILP 会生成 155 条约束和 46 个变量。GLPK 求

$$
\begin{aligned}
&Minimize \quad \tau(\mathfrak{R}, G) \\
&\quad s.t. \\
&\tau(\mathfrak{R}, G) > b_{i,j} * (st_{i,j} + t_{i,j}) \quad (\forall v_i \in V), (\forall d_j \in D); \\
&t_{i,j} > b_{i,j} * cl_{i,j} + \sum_{(v_k, v_i)}^{\forall v_k \in pred(v_i)} (b_{i,j} - b_{k,j}) * C_{(d(v_k), d(v_i))}(T); \\
&st_{k,l} > st_{i,j} + t_{i,j} + (b_{i,j} - 1) * M \quad (\forall (v_i, v_k) \in E), (\forall d_l, d_j \in D); \\
&st_{i,j} > st_{k,j} + t_{k,j} - \psi_{i,k,j} * M \quad \forall d_j \in D \\
&st_{k,j} > st_{i,j} + t_{i,j} - (1 - \psi_{i,k,j}) * M \quad \forall d_j \in D; \\
&\sum_{j}^{\forall d_j \in D} b_{i,j} = 1, \quad \forall v_i \in V; \\
&b_{i,j} \in \{0, 1\}; \\
&\psi_{i,k,j} \in \{0, 1\}.
\end{aligned}
$$

图 3-13 整数线性规划约束总结

Summary of the Integer Linear Program for minimizing computation latency.

解器[187]只需要 0.02 秒就能求出结果。然而，对于有 14 个节点的子图，ILP 会生成 827 条约束和 259 个变量，求解器需要 88.5 秒才能求出结果。此外，Inception-V3[159]模型有 130 个算子，NasNet[37]有超过 700 个算子，一下求解整个 DNN 模型所有的算子是不可行的。因此，本节提出了一个图划分算法和 ILP 求解器协同工作，来减少 ILP 求解器的时间复杂度。

图划分算法需要满足这样的要求：

- 被切分的子图之间依然要保持有向无环的性质，否则子图之间出现环就无法进行调度；
- 被切分的子图的算子数目相对平衡，即切分后形成的两个子图的节点集合 V_0 和 V_1，节点数量 $|V_0|$ 尽可能和 $|V_1|$ 接近；
- 最小化点割集（vertex cut）的数量，来减少子图之间潜在的通信开销。

图划分可以形式化为如下的问题：给定一个 DAG $G = (V, E)$ 和一个超参数不平衡因子 ϵ，找到一个无环的图的二分 $P = \{V_0, V_1\}, (V_0 \subseteq V, \quad V_1 \subseteq V)$ 使得如下的平衡约束能够被满足并且使得点割集中点的数量最小化：

$$
w(V_i) \leq (1 + \epsilon) \frac{\sum_{v \in V} w(v)}{k} \tag{3-7}
$$

本节使用 $\{V_0, V_1\}$ 来表示图 G 的两个子图的节点的集合。设 $w(v)$ 是所有节点的的权值，$w(V_i)$ 是所有属于 V_i 的节点的权值之和。为了能够找到一个合适的切分

策略，本文采用了“升秩（upward-rank）”的概念，其定义如下：

$$uprank(v_j) = 1 + max_{v_i \in pred(v_j)}(uprank(v_i)). \tag{3-8}$$

从直觉上看，v_i 的升秩就是从 DAG 图的源节点（或者 DNN 中的输入节点）到 v_i 的最长的路径的长度。DAG 图中所有节点的升秩的值只需要按照拓扑顺序遍历一遍就可以得到。因此其计算的时间复杂度是 $O(|V| + |E|)$。

基于节点升秩值来将 DAG 图划分为子图是基于 DNN 模型的拓扑结构特征。通过递归的将 DAG 图划分为子图直到所有子图的节点数量 $|V_i|$ 小于 $|NT|$。实验中设置 $|NT|$ 为 12，用户可以根据 DNN 模型和 ILP 求解器的求解时间来设置。

3.6.3.1　基于升秩的图划分算法

算法 1 基于升秩的图划分算法

Input: Directed graph $G = (V, E)$

Result: A feasible partitioning (V_0, V_1) of G

compute upward rank value for each vertex in G

$minUprank$, $maxUprank \leftarrow$ minimum and maximum upward rank value in G

alloc $uprankCount[maxUprank]$,$accUprank[maxUprank]$

for v ***in*** V **do**

    $uprankCount[uprank(v)] += 1$

for $rank \leftarrow minUprank$ ***to*** $maxUprank$ **do**

    $accUprank[rank] = uprankCount[rank] + accUprank[rank - 1]$

$minVertexCutCount \leftarrow |V|$

$minBlance \leftarrow |V|$

$V_0, V_1 \leftarrow \varnothing$

for $rank \leftarrow minUprank$ ***to*** $maxUprank$ **do**

    $w(V_0) \leftarrow accUprank[rank]$

    $w(V_1) \leftarrow |V| - accUprank[rank]$ **if** *balance constraint is satisfied* ***and*** $uprankCount[rank] <= minVertexCutCount$ ***and*** $minBlance < abs(|V_0| - |V_1|)$ **then**

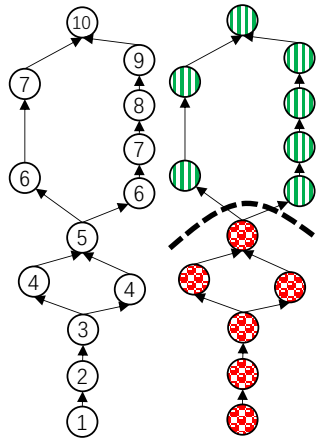
        $V_0 \leftarrow \{v \, \forall v \in V$ **if** $uprank(v) \leq rank\}$ $V_1 \leftarrow \{v \, \forall v \in V$ **if** $uprank(v) > rank\}$

        $minVertexCutCount \leftarrow uprankCount[rank]$

return (V_0, V_1)

算法 15描述了基于升秩进行图划分的伪代码。算法尝试发现所有合适的升秩的值从而能够把图 G 划分为有相近数量的节点，同时最小化点割集中点的数

量，来减少子图之间潜在的通信开销。首先使用公式 3-8计算图中每个节点的升秩的值 (第 1 行)。然后可以获取图中的最大和最小的升秩（第 2 – 3 行）。之后遍历最小和最大的升秩之间的值，尝试把图切分为 2 部分（第 5 – 6 行）。然后计算两个子图的节点数量（第 7 – 8 行）。变量 $accUprank[rank]$ 表示升秩值小于“rank”的节点的数量。之后遍历所有的升秩的值来找到一个能够满足平衡性约束同时点割集中点的数量最小的“rank”值。获取划分的结果（第 9 – 18 行）。这个图划分算法的时间复杂度是 $O(|V|log(|V|))$。在本文中平衡参数设置为 0.2，如果找不到满足两个子图中节点数比较平衡的切分，就减少平衡参数为 0.1 直到找到一个满足约束条件的切分。



(a) Inception-V4 的子图。 (b) 一个合适的切分。

图 3-14 (a)Inception-V4 的一个子图的升秩，(b) 一个满足约束的切分。

[Graph partitioning example with up rank] (a)An example graph from Inception-V4 with uprank，(b)An acyclic partitioning.

图 3-14展示了使用升秩进行图划分的例子。图中圆圈内的数值即为升秩。图中有 6 个节点的升秩的值是小于 5，有 7 个节点的升秩值大于 5。可以发现“rank”值为 5 是一个合适的阈值。因为 $(1+0.2)\frac{13}{2}=7.8$，而两个子图的节点数 $6<7.8$ 且 $7<7.8$，平衡的约束能够满足；同时点割集中点的数量是是 1，达到最小化。同时，两个子图 V_0（红色点状填充）和 V_1（绿色线状填充）之间也是有向无环的。

3.6.3.2 切分长短期记忆网络（Long Short Time Memory (LSTM) Network）

LSTM[6] 是一种循环神经网络（Recurrent Neural Network (RNN)），在语音识别和自然语言处理中被广泛的使用。通常 LSTM 的一个单元有 4 个门包括输入调制门、输入门、遗忘门和输出门。四个门的计算互相独立。HOPE 以每个门为一个单位进行调度。

3.6.4　启发式的调度算法

本节提出一个启发式的调度算法，以很快地获得一个较优的执行计划 Re。解决方案的关键是以批处理的方式来调度节点，在这一批内的节点其获得的是最优解（对整个图来看是局部最优解)。此外，算法会先调度可以开始执行（即其所有的前驱节点都完成) 时间最早的节点。节点的可以开始执行时间的定义是其前驱节点完成时间的最大值。输入节点的开始执行时间设置为 0。

3.6.4.1　贪心的启发式算法

算法 2 基于贪心策略的搜索算法

Input: DAG with profile data; computing device

Result: Execution plan

16 $opsQueue \leftarrow inputNodes$

17 **while** $opsQueue$!=∅ **do**

18 　sort opsQueue by starting execution time $kOps \leftarrow opsQueue.getTopK()$

19 　$searchMiniEndTime(kOps, devices)$

20 　$updateSuccessorStartingExecutionTime(kOps)$

21 　$opsQueue \leftarrow kOps.successors$

22 **end**

图 22展示了启发式算法的伪代码. 首先，算法贪心的选择 K 个开始执行时间最早的节点（通过一个简单的快速排序即可，在伪代码的第 3 到 4 行)。然后这 K 个节点等待调度。之后，算法通过一个搜索树枚举所有可能的节点到设备的映射，并且选择其中完成时间最小的方案。如果一个 DAG 有 y 个节点，目标的平台有 x 个设备，启发式算法的时间复杂度是 $O(\frac{y}{k} * x^k)$。在本文中，对于 x 为 2 的设置 K 为 4，对于 x 为 3 的设置 K 为 3（第 5 行)。最后，根据这 K 个算子的调度结果更新后继节点的可以开始执行时间（第 6 行)。

3.6.4.2　启发式算法中的节点合并

在 DNN 模型中，存在有许多运行时间比较短的算子，和卷积等算子相比，其计算复杂度很低。例如，Inception-V3 模型在 Redmi 平台上，*ReLU* 算子执行时间小于 0.1 毫秒，但是其中一个 *Conv2D* 算子的执行时间超过 3 毫秒。这个观察启发本文引入了算子合并的算法。对于执行时间较短的算子，如果他只有一个前驱，HOPE 把它和其的前驱算子映射到同一个计算设备上。合并后的节点称为 *super-op*。HOPE 首先运行算子合并，把符合要求的节点合并为一个节点，之后再执行上述的启发式调度算法。对于支持算子融合的 DNN 框架，调度器会把融合后的算子作为一个节点进行调度。

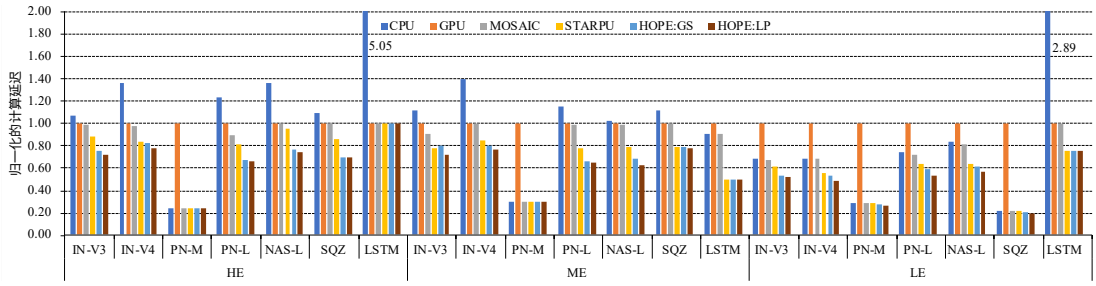


图 3-15 三个移动平台上归一化的推理延迟

Figure 3-15 Normalized inference latency on the three mobile systems.

3.6.5 实验

移动平台：本文的实验设备 3 个商业消费者使用较多的的移动平台，包括 Redmi Note4x（低端平台，配备 Snapdragon 625，标记为 **LE**）Xiaomi 9SE（中端平台，配备 Snapdragon 712, **ME**），HUAWEI P40（高端平台，配备 Kirin 990, **HE**）。实验平台涵盖了低端平台（市场价格千元左右）、终端平台（市场价格两千元左右）和高端平台（市场价格三千元及以上）。

DNNs 模型和数据集: 本实验在 7 个最被广泛使用的 DNN 模型上，包括 Inception-v3[159]（**IN-V3**）、Inception-v4[188]（**IN-V4**）、PNASNET-mobile[4]（**PN-M**）、PNASNET-large[4]（**PN-L**）、NASNET-large[37]（**NAS-L**）、SqueezeNet[35]（**SQZ**）、和 LSTM[6] 来衡量 HOPE 的性能。LSTM 按照 DeepSpeech[7] 的配置来衡量。所有的卷积神经网络（Convolution Neural Network, CNN）网络都在 ImageNet[162] 数据集上训练。LSTM 包含一层，有 1024 个隐藏单元和 10 个时间步。本实验衡量的 DNN 模型既包括了为移动平台高度优化的模型（**PN-M** 和 **SQZ**），也包括为了追求最高的准确率而训练的较大的模型（**IN-V4** and **PN-L**）。此外，DNN 模型在算子数量方面也展现出了巨大的差异，从 40 个算子到 1049 个算子都有。如此数量巨大的算子对于调度算法提出了很高的要求。

实现: HOPE 使用最新版本的 GNU Linear Programming Kit (GLPK) 来求解 ILP 的表达式。异构并行的执行引擎基于阿里巴巴开源的 MNN 框架实现，增加了约 3 千行 C++ 的代码。图划分的模块和调度的模块使用 Python 语言实现，包含约 4 千行 Python 代码。

3.6.5.1 总体性能

本实验用不同的硬件平台和配置来衡量 HOPE 的有效性，并使用当前最好的异构执行引擎 MOSIAC，以及当前最经典使用最广泛的启发式调度器 StarPU[189] 的 dequeue-model-data-aware-ready （DMDAR）策略作为对比对象。

本文在 **LE** 平台上使用 4 个 CPU 核心，在 **ME** 和 **HE** 上使用 2 个大核心，以及所有平台上的 GPU 来衡量 HOPE 的性能。注意 HOPE 在同时使用 CPU 和 GPU 的时候并不使用 CPU 上所有的大核心和小核心。这是因为框架并不能感知到底层的 CPU 不同大小核心的区别，因此会把工作负载均匀的分配到所有的

CPU 核心上。由于小核心的性能非常弱，如果没有精巧的调度的话会反而拖慢整体的 DNN 推理的工作负载。而 HOPE 能够感知到大小核心并进行精巧的调度。章节 3.6.5.4中展示了详细的实验结果。

图 3-15展示了在 3 个平台上 6 个不同版本的归一化的推理延迟。对于每个 DNN 模型，本实验衡量 6 个不同的版本，分别是只用 CPU（**CPU**），只用 GPU（**GPU**），MOSAIC 使用 CPU 和 GPU（**MOSAIC**），StarPU's (DMDAR) 策略使用 CPU 和 GPU（**STARPU**），HOPE 的启发式调度器（**HOPE:GS**），和 HOPE 的基于整数线性规划的调度器（**HOPE:LP**）。

图 3-15表明了 HOPE ILP 和启发式调度器的有效性。首先 HOPE 的性能有效的超过了 MOSAIC。具体而言，HOPE 使用 ILP 求解器的延迟在这 7 个模型中，三个平台求平均，比 MOSAIC 分别低 23.4%、24.1%、2.9%、29.4%、31.1%、21.3% 和 23.3%。这是因为 HOPE 能够把算子映射到不同的计算设备上并行的运行，而 MOSAIC 只能在不同的设备上串行地执行。第二，HOPE ILP 版本的调度器 **HOPE:LP** 比 StarPU 的调度器的延迟要分别少 13.7%、9.1%、2.9%、17.9%、18.0%、10.0% 和 0%。这是因为对于每个子图，基于 ILP 的求解器能够找到最优解，因而相比启发式的 StarPU 性能有明显提升。第三，HOPE 的启发式算法 **HOPE:GS** 相比 **STARPU** 也能够明显的减少推理延迟，在 7 个模型上分别减少 8.2%、3.2%、0.5%、12.9%、11.7%、0.0% 和 0.0%。性能的增益来自于两部分，第一部分是在调度之前 **HOPE:GS** 先将算子进行了合并因而降低了潜在的通信开销；第二部分是 **HOPE:GS** 在调度算子时拥有更大的搜索窗口（同时考虑 K 个节点），而 StarPU 则只考虑一个节点。StarPU 相比，LSTM 的性能并没有提升，这是因为 LSTM 的每个 cell 只有 4 个门（算子），调度的空间很小。实验结果清楚的表明, 与 **MOSAIC** 和 **STARPU** 相比，HOPE 能够有效的降低 DNN 端到端的推理延迟。

HOPE 在三个平台上的 **PNAS-M** 都展现出了和只使用 CPU 相似的结果。这是因为 **PNAS-M** 是特殊的为 CPU 推理优化的。比如说，在 **HE** 上，**PNAS-M** 在 CPU 上需要 43 毫秒而在 GPU 上需要 183 毫秒，因而 GPU 的参与并不能有效的降低端到端的执行时间。

3.6.5.2　CPU 性能的可拓展性

本小节考虑 CPU 和 GPU 之间的相对峰值性能改变时 HOPE 的可拓展性[19]。通常，CPU 的频率和性能成正比。实验中通过改变 CPU 的频率来改变 CPU 的峰值性能。本实验使用 **LE** 平台，其 CPU 频率的变化范围是从 652MHz 到 2016MHz，使用 CPU 的 *scaling_avaliable_frequencies* 列表中最高、中间、最低三个档位的 CPU 频率来衡量。图 3-16展示了在 DNN 模型中归一化的延迟。**HOPE:LP** 在最低频率、中等频率、最高频率分别比 **MOSAIC** 减少至多 26.4% （平均 19.2%），25.2% （平均 17.5%）和 29.1% （平均 22.1%）推理延迟；比 **STARPU** 减少至多 17.2%（平均 10.8%），20.4%（平均 9.3%）和 17.0%（平均 10.3%。在 **HOPE:GS**

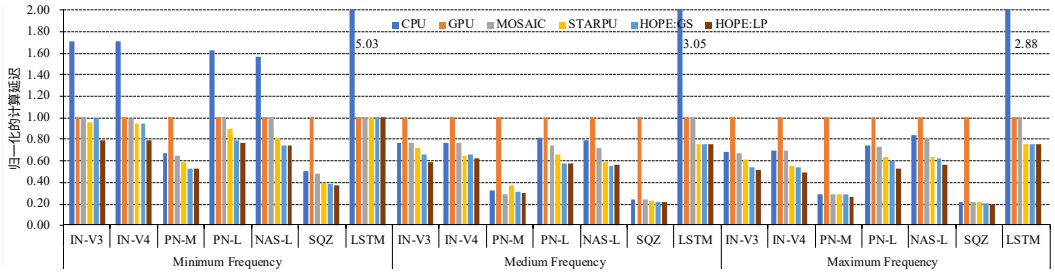


图 3-16 在 LE 平台上随 CPU 频率变化的归一化的推理性能

Figure 3-16 Normalized inference latency with CPU frequency scaling on LE

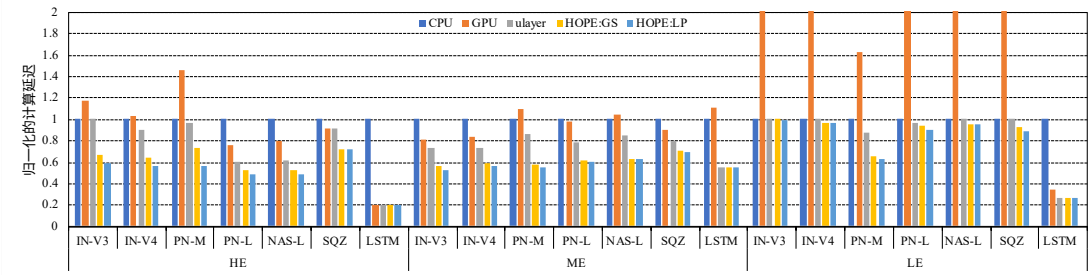


图 3-17 与 μLayer 的性能对比

Figure 3-17 comparison with μLayer

上也能够观察到相似的结果。实验结果表明当 CPU 和 GPU 峰值性能的比例变化之后，HOPE 性能仍然明显超越 **MOSAIC** 和 **STARPU**。

3.6.5.3 与单层加速的工作对比

Kim[48] 提出了 μLayer，可以使用移动平台上的异构计算芯片同时执行一个算子的计算。μLayer 采取了三种优化的策略，包括单层网络的协同执行，对处理器有好的量化和分支并行。本实验的目标是针对全精度（FP32）模型的推理，因此 HOPE 和 μLayer 对比的时候也是对比全精度。μLayer 并没有开放源代码，因此本文按照 μLayer 的文章[48] 描述的基于 ARM Compute Library(ACL)[55] 实现了其运行时。本文也在 ACL 上实现了 HOPE 的运行时系统。ACL 相比 MNN 存在的一个好处是，ACL 的基于 ARM Neon 的 CPU 后端实现和基于 OpenCL 的实现使用了相同的数据布局。因而在 CPU 和 GPU 之间传输数据的时候不需要额外的数据布局转换的开销。对于 CPU 和 GPU 使用不同数据布局的 DNN 框架，μLayer 会在每一层都引入额外的数据布局转换的开销。本实验使用和上一节 3.6.5.1相同的软硬件实验配置。图3-17展示了 μLayer 和 HOPE 的两个调度算法的归一化的执行延迟。结果表明，**HOPE:GS** 能够比 μLayer 减少 32.9%（**HE**）、32.8%（**ME**）和 25.4%（**LW**）的延迟。**HOPE:LP** 能够减少最多 40.6% （**HE**），35.9%（**ME**）和 28.0%（**LW**）。原因是 μLayer 的单层异构加速模型每一个算子都需要 CPU 和 GPU 之间进行细粒度的同步。注意和 μLayer 相比 LSTM 的推理延迟并没有降低原因是 μLayer 和 HOPE 采取了同样的划分策略。HOPE 在 **HE** 和 **ME** 能够分别减少平均 25.8% and 21.3% 的延迟，但是在 **LE** 上平均只有 8.0%。

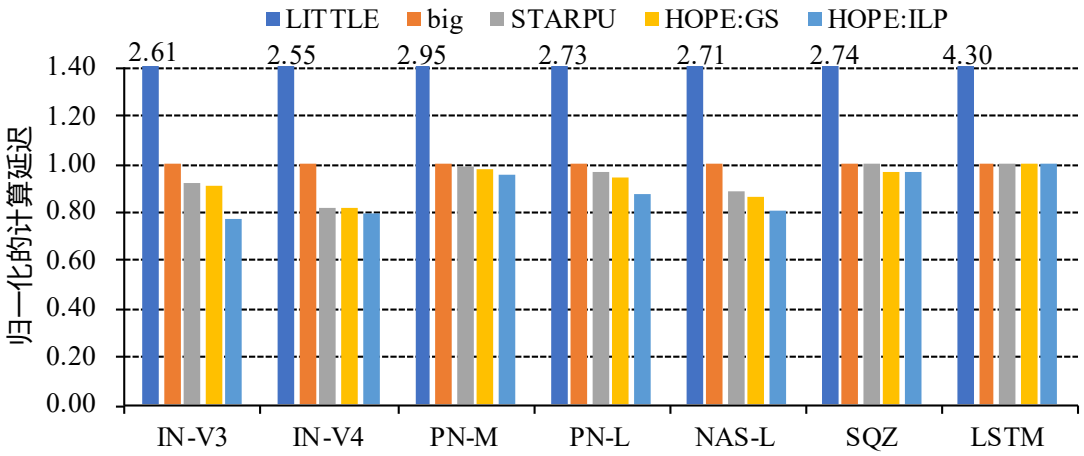


图 3-18 在大小核上归一化的推理延迟

Figure 3-18 Normalized latency on the big.LITTLE cores

原因是在 **LE** 上 CPU 和 GPU 的性能非常的不平衡。综上所述，HOPE 的性能仍然要明显的优于单层执行引擎 μLayer 。

3.6.5.4　CPU 大小核心架构（big.LITTLE）

移动设备的片上系统（System on Chip，SoC）普遍采用了 ARM 的大小核（big.LITTLE）的技术来平衡性能和功耗。通常小核心处理器用来实现更高的性能功耗比而大核心通常用来实现最高的计算性能。HOPE 可以感知到 SoC 上的大小核心并且使用大小核协同的来调度 DNN 的模型。本实验在 **HE** 平台上使用 2 个大核心和 4 个小核心来衡量 HOPE 的有效性（**HE** 上有 2 个大核心、2 个中核心、4 个小核心）。不像使用 CPU+GPU 的模式，使用大小核心协同推理时通信开销可以被忽略。因为大小核心共享同样的内存空间，使用同样的数据布局。图 3-18展示了实验结果。具体而言，HOPE 相比 **STARPU** 能够减少 15.7%、2.2%、2.6%、9.8%、8.3%、3.2% 和 0% 的推理延迟。当只使用 CPU 的时候 HOPE 能够减少推理延迟，提高用户体验。由于 CPU 大小核心之间没有通信开销，**HOPE:GS** 的表现就和 **STARPU** 差不多，因为 **HOPE:GS** 并不能从合并算子中受益。

3.6.5.5　讨论

开销分析。图3-19展示了 HOPE 的 ILP 调度器和启发式调度器的开销。实验结果表明 ILP 的求解器引入了 1.4 到 4.5 秒的开销，启发式调度器引入了大约 0.89 到 1.0 秒的开销。我们的启发式调度器引入了和 StartPU 的 DMDAR 策略相似的开销。但是，由于我们的执行计划是在运行模型之前静态做的，并且只需要运行一次即可，所以开销可以被忽略。我们也衡量了图预处理对调度结果和性能的影响。如果不把算子合并为 *super-op*, 基于 ILP 的调度器分别花了 68 秒和 645 秒调度 Inception-V3 和 Inception-V4，但是与合并 *super-op* 的性能相近；具体而言性能差距分别是有 1.6% 和 2.7%. 注意，如果不把 PNASNET-large, PNASNET-

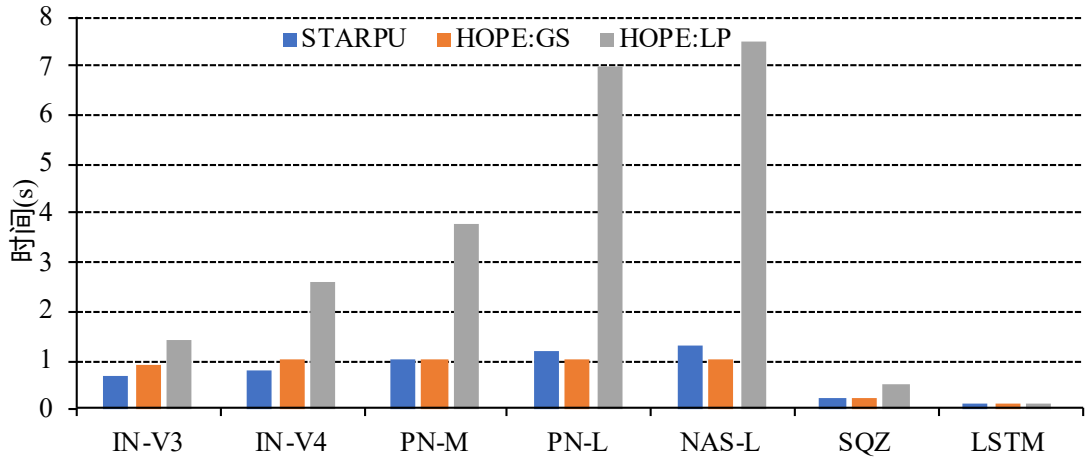
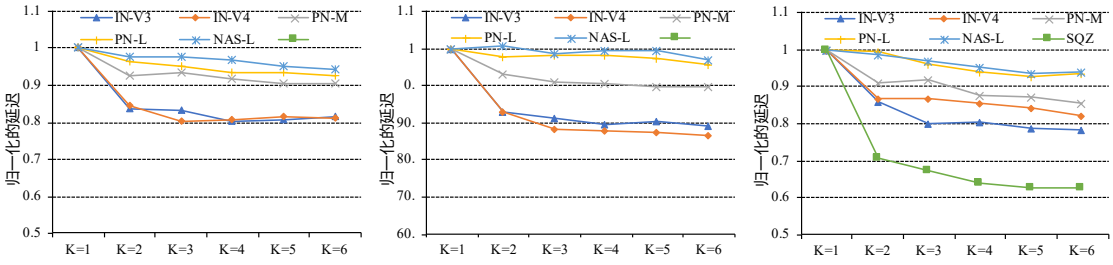


图 3-19 调度器的开销

Figure 3-19 The Overhead of scheduler



(a) HE **(b) ME** **(c) LE**

图 3-20 启发式调度器中 *K* 值的选择对性能的影响

Figure 3-20 The impact of the choice of the value of *K* in the heuristic scheduler on performance

mobile and NASNET-large 的算子合并为 *super-op*，则需要花费超过 24 小时的时间也难以求解。因此我们忽略了性能对比。

启发式算法中 *K* 值的影响。启发式算法选择前 K 个有最小化执行时间的算子，并且以 ***K*** 个算子为一个批次进行调度。为了分析 ***K*** 的选择对最终调度结果的影响，本实验研究了当 ***K*** 从 1 增加 6 时端到端延迟的变化。注意计算延迟是离线的调度器所估计的时间。***K*** 等于 1 意味着调度器一个一个的调度 DNN 的算子。绿色的方块标记的点代表 **SQZ**，在 **HE** 平台上，当 ***K*** 从 1 增加到 2 计算延迟从 1.0 降低到 0.64。由于 LSTM 只有 4 个门因而我们忽略了它的结果。结果表明当 ***K*** 增加的时候总体的计算延迟是降低的。但是当 ***K*** 只大于 4 之后整体的性能收益就很低了。因而我们选择 4 作为 ***K*** 的值。

3.7 本章小结

本章主要解决现有的基准测试框架不能刻画 DNN 在多种软硬件配置下性能特征问题，并提出了一个支持移动端和云端的基准测试和自动调优框架

DNNTune，在多种硬件类别、硬件等级、软件配置、编程框架和 DNN 优化算法上刻画了 13 个 DNN 模型的性能特征，得到了 5 个核心观察。实验观察发现，（1）没有一个软件框架对于所有的 DNN 模型都能达到性能和功耗的最优，需要针对不同的目标自动衡量并选择最佳配置；（2）没有一个计算单元对所有 DNN 模型都能达到性能和功耗的最优，需要将 DNN 模型映射到不同的硬件上执行。基于核心观察，本章提出了基于端云协同的自动调优和基于多设备异构并行的调优两种调优方法。这两种方法能够将 DNN 模型自动映射到多种计算平台和设备执行，明显降低 DNN 模型推理的延迟和功耗。

第 4 章　基于全局分析和局部变换的深度学习模型编译优化

4.1　相关背景与研究动机

在第 3章，我们通过将 DNN 算子映射到不同的计算平台来提升推理的性能，深度学习框架依赖于编译器和算子库运行在智能处理器上，并达到较高的性能。然而，要想进一步的提降低深度学习应用的推理延迟，必须要在编程框架的更下层，即编译器的层级为 DNN 模型生成更高性能的代码。本章通过分析 DNN 模型多层级的数据复用信息和依赖关系，来为智能处理器生成高性能的代码，进一步优化深度学习应用的性能表现。

程序员通常使用像 TensorFlow[38] 和 PyTorch[140] 这样的深度学习编程框架来编写一个计算图。计算图描述了数据如何流过不同的算子（Operator），其中节点代表算子，例如 *add*, *pooling* 和 *softmax*，而边代表算子之间的数据（张量）依赖关系。深度学习框架通常提供高层级简单易用的编程 API，隐藏了底层复杂的面向多种硬件平台的代码实现，从而使得 AI 的开发者可以快速地开发 DNN 模型。然而，高层级算子抽象同时也为底层的性能优化带来了极大的挑战。在生产环境中推理预训练好的模型时，其响应时间非常重要[25,190]，性能问题尤其显著和关键。

为了提供高性能的算子实现，研究人员尝试通过手写高性能的算子计算核（kernel）或者基于自动调优技术（例如 auto-tuning[63] 和 auto-scheduling[127,191,192]）使用编译器生成 kernel。优化单个算子的性能很重要，然而生成的 kernel 之间仍然存在着上层算子的界限，从而限制了进一步挖掘算子性能的机会。

当前的 DNN 模型的架构由许多不同的算子组成。算子之间存在着许多算子间并行（inter-operator parallelism）和冗余的计算。编译器如果不从整个 DNN 模型架构的视图分析，会陷入如下的困境：（1）编译器单独地优化每个算子实现，不能跨越算子之间的边界分析数据流和指令流；（2）不知道两个 kernel 的哪些指令可以被重排序并被整体的调度，从而发掘硬件的并行性或者消除冗余计算；（3）不能够发掘哪些张量的缓存能够在两个算子之间安全地复用，从而避免代价高昂的内存访问；（4）不能从全计算图获取信息。因而，算子之间的边界阻碍了编译器的整体分析，成为提升整体推理性能的主要的障碍。这其中蕴含着许多优化的机会。

通过把多个算子合并到一个 kernel 中，算子融合（Kernel Fusion）能够打破算子之间优化的界限，特别是在推理场景中能够大幅提升 DNN 推理的性能 [118]。这些研究工作的方法可以分为基于模式匹配[50]，基于手写规则[118]，基于循环分析[121] 或者即时编译（Just-in-time Compilation）[120]。算子融合是一个很好的跨算子边界的优化方法。

算子融合需要把一个大的复杂的计算图切分为多个小的由一个或者多个算子组成的子图，这样才能够有效地在每个子图上进行代码分析和优化。计算图

的切分是算子融合的关键，然而发现正确的（或性能优异）的计算图切分的边界是非平凡的。把算子划分到错误的子图将会导致额外的通信开销，从而丧失潜在的优化机会。如果简单地把尽可能多的算子融合为一个 kernel，会降低算子的算子间并行性，并增加 GPU 的缓存压力。现有的工作通过增加手工的模板和启发式规则进行图的切分，但是这样的方法难以适应多种多样的 DNN 计算图的结构。就像在后文中展示的，当前最优的算子融合的方法仍然可能会错过许多优化的机会，有很大的提升 DNN 推理性能的空间。此外，目前的算子融合的方法高度受限于框架所支持的算子，融合后的算子通常也需要框架支持。这样的缺陷极大地限制了编程框架和编译器的可拓展性。

本章提出了 Souffle ，一个更好的支持跨 DNN 算子边界的面向推理优化的编译器。本文的核心观察是：（1）要发掘更多的优化机会，需要在整个 DNN 的张量计算图上进行全局的依赖分析（而不是单独的算子上），（2）通过张量表达式（Tensor Expression）[51] 来构建张量元素之间的依赖关系和表达算子的计算模式。从张量的依赖图中，Souffle 可以获取到例如张量的形状、跨越算子边界的张量的活跃区间以及推断张量表达式输入和输出张量之间元素级的依赖关系。Souffle 先在计算图中进行全局的依赖分析，之后进行精心设计的启发式策略，有效的把张量表达式划分为一系列的子程序（Sub-program），以最大化指令级优化和数据复用的机会。在子程序中，Souffle 进行一系列语义不变的数学变换来减少计算。Souffle 通过综合考虑子程序内张量表达式的计算特征来优化子程序的调度（Schedule）。通过跨算子边界的多层次的依赖分析，Souffle 能够有效的跨算子边界地优化一个子程序，以支持调度存取指令（Load/Store Instructions）提高指令级并行，和复用高速缓存上的张量以减少内存访问开销。张量表达式是在整个计算图中全局地分析和优化，而不是在单独的算子上。算子之间的边界自然就被消除了。

Souffle 提供了一系列的分析、公式和算法来跨算子边界的发现和定位优化机会，发掘和利用了之前的框架没有发现的新优化机会。Souffle 非常的高效，具有很好的可拓展性，能够融合当前最先进的深度学习编译器不能融合的算子。Souffle 能够读取 TensorFlow 的模型，并转换为由张量表达式构成的程序。Souffle 可以集成到通用的深度学习编译器（例如 TVM[51,136]）来提升端到端的 DNN 推理性能。

本工作基于 TVM 实现了 Souffle 的原型，在六个典型架构的 DNN 的模型上衡量了 Souffle 的性能，并且和四个当前最先进的深度学习算子融合的框架（包括 Ansor[136], TensorRT[131], Rammer [62] 和 Apollo[121]）做了性能对比。在 NVIDIA A100 GPU 上的实验结果表明，Souffle 能够显著的优于目前的解决方案，与厂商优化的推理库 TensorRT 相比，可达到最高 5.17× 的加速比。

本章的研究工作主要有以下的贡献：

- 提出了新的跨算子边界探索和发现优化机会的方法；
- 在 kernel 层展示了全局分析局部变换的有效性，弥补了目前算子融合方

法的缺陷；

- 证明了全局分析能够有效地跨越算子边界来优化性能。

4.2　研究动机：以 BERT 为例

4.2.1　示例模型简介

本节以在 NVIDIA GPU 优化 BERT[10] 模型为例来展示当前最先进的工作和 Souffle 优化效果。BERT 模型基于 Transformer[9,11] 架构，是目前许多最好的 DNN 模型的基础架构。在本节的实验使用了从 TensorRT 中获取的 BERT 模型，数值精度为 16 比特浮点数（FP16）。

图 4-1(a)(b) 分别描述了 TensorRT 和 Apollo[121] 是如何把 BERT 模型中的算子映射到 kernel 中的 [1]。这个子图中包括了典型的 DNN 算子，例如矩阵乘（General Matrix Multiplication，*GEMM*）、张量变维（*Reshape*）；元素级的算术运算，包括“加算子”（*Add*）、“乘算子”（*Mul*）和规约算子，例如“规约加”（*Reduce_sum*）。如果编译器把每个算子都生成为独立的 kernel，将会对推理的性能有很大的影响。

4.2.2　性能评估

本实验使用 NVIDIA Nsight Compute 性能收集器来测量各个编译器生成的 kernel 在 NVIDIA A100 GPU 上的性能。表 4-1展示了 TensorRT 和 Apollo 都不能为 BERT 的模型架构提供一个性能优异的算子实现。TensorRT 和 Apollo 对于图 4-1中生成的 kernel 分别需要访问 26.25（兆字节，MB）和 27.78 MB 的 GPU 的全局内存（Global memory），执行时间分别是 143.8 μs 和 179.1 μs。而 Souffle 提供了一个更好的映射的策略，即把整个子图都映射并生成为一个 kernel。这个策略极大地减少了从全局内存访存的数量（减少为 8.87M），执行时间减少为 57.7 μs。与 TensorRT 和 Apollo 相比，Souffle 分别达到了 2.5× 和 3.1× 的加速比。

4.2.3　错失的优化机会

经过仔细的分析性能收集的结果和 kernel 的映射，实验发现了几个 TensorRT 和 Apollo 没有发现的优化机会，在下文详细说明。

不能探索计算密集型和访存密集型算子之间的优化机会。如图 4-1中所示，BERT 中一部分的算子执行按元素的内存操作，例如 *Reshape* 和 *Permutation*。TensorRT 和 Apollo 都使用手工写的规则来融合相邻的按元素操作内存的算子。然而，他们都不能够有效地和相邻的算子融合，例如 *GEMM* 算子。Souffle 能够在计算密集和访存密集的算子之间做融合，最终消除了所有的访存密集的算子。TensorRT 相比，Souffle 最终减少了 84.4μs 端到端的延迟。总结而言，手工编写的规则不大可能覆盖 DNN 模型中所有的算子组合和计算的模式，因而会错失这个例子中

[1] 为了图的清晰和可读性，数据布局转换的 kernel 没有被画到图中。

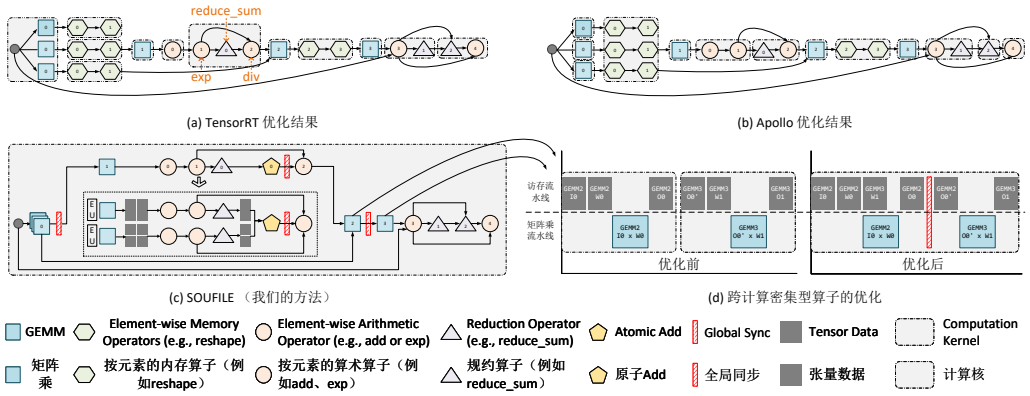


图 4-1 TensorRT (a), Apollo (b) 和 Souffle (c) BERT 的 Attention 模块算子融合的结果

Figure 4-1 How TensorRT (a), Apollo (b) and Souffle (c) map BERT into kernels

表 4-1 图4-1中生成的 kernel 的性能特征

Table 4-1 Performance for the generated kernels of Figure 4-1

	TensorRT	Apollo	Souffle
总执行延迟（微妙 μs）	143.81	179.07	57.73
–计算密集型 kernels 延迟	43.49	61.1	41.7721
–访存密集型 kernels 延迟	100.32	117.97	15.9579
kernel 的数量	17	14	1
从设备内存访问的字节数 (M)	26.25	27.78	8.87

的优化机会。

不能为规约算子生成最佳融合策略。图 4-1(a) 和 (b) 所示，TensorRT 和 Apollo 的融合策略都选择把 *GEMM* 和规约算子（例如 Reduce_sum）分别映射到不同的 kernel，这是因为 *GEMM* 产生的张量需要先存回全局内存，然后再由规约算子从全局内存加载到片上的共享内存（Shared Memory），这就带来了高昂的访存开销。而 SOUFFLE 激进地把规约算子（例如图 4-1中的 R0、R1、R2 算子）与相邻的计算密集的算子（例如 *GEMM0* 和 *GEMM1*）融合到一起。SOUFFLE 通过两阶段的方式把计算密集算子与规约算子融合到一起：首先在每个线程块（Block）内部进行局部的规约求和，然后使用 *atomicAdd* 原子指令进行全局规约求和，这样整个张量的数据都保存在片上，只有局部规约的结果需要写回全局内存。注意全局同步（*global synchronization*）需要在所有活跃的 block 之间进行同步。这个优化对图 4-1中所有的规约算子都适用。此外，SOUFFLE 能够把算术运算算子 1（arithmetic operator 1）的计算结果缓存在片上，在算子 2（arithmetic operator 2）的计算中复用。这个融合策略与 TensorRT 相比降低了 78.5% 的内存读写（从 8.55MB 到 4.74MB），带来了 18.05μs 的延迟降低。

不能探索计算密集型算子之间的优化机会。和许多其他的 DNN 框架一样，TensorRT 和 Apollo 能够把计算完全相同（只有权值不同）的算子融合，但是不能

```
rk=te.reduce_axis((0,768), name='rk')
A=te.placeholder((384,768), name='A')
B=te.placeholder((768,768), name='B')
C=te.compute((384,768), lambda i,j te.sum(A[i,rk]*B[rk, j], axis=[rk]))

```

图 4-2 BERT 中 *GEMM* 算子的 TE 的实现

Figure 4-2 A TE example for the BERT *GEMM* operator

够跨算子边界探索计算密集型算子的优化机会。考虑如图 4-1(d) 所示的两个独立的 *GEMM* 的算子，这两个算子分别执行从全局内存到片上共享内存（Shared Memory）[193] 的拷贝，然后执行张量计算核心（Tensor Core）的计算。这两个 *GEMM* 算子的计算到 GPU 的流处理器（Streaming Multiprocessor, SM）的映射是不同的。第一个把两个 *GEMM* 的运算映射到了两个 kernel，这两个算子之间还有其他的算子，且目前的工作都认为两个 *GEMM* 算子是不能融合到一个 kernel 的。而第二个把两个 *GEMM* 都融合到了一个 kernel。第二个 *GEMM* 从全局内存到共享内存取数的操作可以和第一个 *GEMM* 的张量计算核心计算的操作在指令级并行运行。TensorRT 和 Apollo 都是前一种方式，而 Souffle 采用后一种方式。如图 4-1(d) 所示，把两个 *GEMM* 映射到两个 kernel 带来了较低的 GPU 利用率。因为两个 *GEMM*（*GEMM2* 和 *GEMM3*）和它们的输入（I0,I1）以及权值（W0,W1）都是先从全局内存加载到共享内存，然后再进行张量计算核心的计算（HMMA）。因而，在 HMMA 的流水线工作的时候访存的流水线会空闲。图 4-1(d) 中展示了利用了指令级并行性的更优的实现。GEMM 的权值 W2 可以和 GEMM1 的 HMMA 运算并行执行。Souffle 支持这样的优化。

4.2.4　核心想法

根据以上的观察，从全局的视图分析 DNN 模型，从数学上等价变换算子，并进行算子之间的联合优化是非常重要的方法。从 DNN 架构的全局视角能够获取到重要的张量的信息，例如活跃区间、张量的大小等；之后通过元素级别的细粒度的分析来进行 kernel 优化。本章提出的 Souffle 可以打破目前 DNN 算子融合的鸿沟以提高 DNN 推理的性能。

4.3　术语和背景知识

Souffle 利用 TVM 的张量表达式（Tensor Expression，TE）[51] 作为中间表示来进行分析和优化。TE 描述了每个输出张量的元素是如何被输入张量的每个元素计算出来的。图 4-2展示了一个 BERT 模型中的 *GEMM* 算子的 TE 表示。其中，*rk* 定义了规约的维度 (在张量的哪个维度进行规约操作)，其规约的范围是从 0 到 768。然后，它定义了两个输入的占位符张量。最后，输出张量 *C* 由 *compute*

函数产生，*compute* 函数定义了其输出的形状和要产生 C 的一个元素所需要的计算；迭代变量 i 和 j 分别和输出张量的维度相对应，其迭代的区间为 C 的两个维度的长度（分别为 384 和 768），可以很自然地被推断出来。本质上来讲，TE 以纯函数式语言来表述了张量的计算，每个输出元素都可以被独立地计算出来。更多详细的内容可以参见 TVM 的官方网站[194]。需要注意的是，本章的优化方法可以被应用在其他的深度学习的领域特定的语言（Domain-Specific-Language）上，例如 antares[195] 和 Tensor Comprehension[129]。本工作选择 TVM，因为它是目前最流行的深度学习编译器，并且具有完善的工具链支持。

4.4 框架设计

Souffle 是一个 DNN 模型优化的框架。Souffle 打破了章节 4.2.1中讨论的目前 DNN 优化框架的局限，旨在提高数据复用，为规约操作提供更好的优化，并进行跨算子边界的优化。当前 Souffle 的实现主要关注于单个 GPU 上的推理优化，支持 TensorFlow 的模型和 NVIDIA 的 GPU。但是 Souffle 中实现的许多分析和优化的方法同样适用于 PyTorch 和其他的智能处理器。图 4-3展示了 Souffle 整体的架构，其输入为 TensorFlow 的模型，使用 TVM 把模型下降（Lowering）为 TE ，之后在 TE 上进行分析和变换。Souffle 的工作流程如下：

TE 下降。对于一个 DNN 的模型，Souffle 首先把每个算子下降为对应的 TE ，形成一个由 TE 组成的程序，张量之间的依赖关系构成了一个依赖图。

全局的计算图分析。下降后的 TE 程序被传递到 Souffle 的分析模块。Souffle 在张量依赖图上进行两个层级的依赖分析。在张量层，Souffle 抽取张量的形状、活跃区间、计算访存比和每个张量在计算图中被使用的次数等信息。在张量中的元素层，Souffle 分析输出张量和输入张量的元素间的依赖关系。

TE 程序的切分和 TE 的变换。 Souffle 进行全局的分析，来确定正确的用来划分 TE 程序的算子，并以这些算子为割点，将 TE 程序划分为多个子图，每个子图对应生成一个 kernel。子图被和张量的数据流分析结果被送到 TE 转换模块来生成数学等价但是更优化的 TE 。具体的 TE 变换将在下一节介绍。

调度生成、联合优化和代码生成。被转换过的 TE 子程序之后被送到调度生成器中（例如 Ansor[136] 和 Roller[65] ）中来为 TE 程序生成一个调度。之后，Souffle 把一个子程序中的调度合并到一个统一的调度，进而在子程序内进行数据复用以及指令联合优化。这些优化需要进行跨算子边界的指令重排序、插入同步指令等。最后，被优化过的子程序被送到后端生成 CUDA kernel 和可执行的二进制文件。

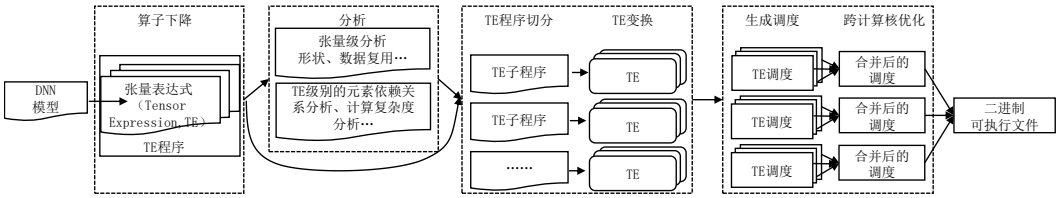


图 4-3 Souffle 框架的整体设计

Figure 4-3 The overall workflow and architecture of Souffle

```
A_exp=te.compute((384,768),lambda i, j:tir.exp(A[i,j]))
A_sum=te.compute((384,),lambda i:te.sum(A_exp[i,rk], axis=[rk]))
A_softmax=te.compute((384,768),lambda i, j:A_exp[i,j]/A_sum[i])

```

图 4-4 BERT 中 *Softmax* 算子下降为多个 TEs

Figure 4-4 Lowering BERT *Softmax* to multiple TEs

4.5　实现

SOUFFLE 的核心是基于下降后的 TE 进行一系列用来分析发现和确定优化机会的方法（章节 4.5.3），和一系列减少计算开销的变换（章节 4.5.6）。

4.5.1　代码实现

SOUFFLE 基于 TVM，用大约一万行 C++ 的代码和一千行的 Python 代码实现了 SOUFFLE 。SOUFFLE 使用 Ansor[136] 来为 TE 生成调度，其他的 TE 调度器（例如 Roller[65]）也可以被集成到 SOUFFLE 中。在一个 TE 子程序中，SOUFFLE 可以基于 TVM 生成的 TIR 把 TE 子程序内的 TE 的调度合并到一起，并进行联合的优化。之后 SOUFFLE 基于 TVM 的后端来生成 CUDA 代码。

4.5.2　张量表达式下降

对于一个输入的 DNN 模型，SOUFFLE 首先把静态的模型转换为一个 TE 程序。SOUFFLE 利用 TVM 的 Relay 模块来把 DNN 中的算子下降为对应的 TE 。注意如图 4-4所示，某些算子（例如 *Softmax* 和 *LayerNorm*）可能会被下降为多个张量表达式 TE 。通过算子下降这一步，SOUFFLE 把输入的 DNN 模型转为了一个有向无环计算图（张量依赖图），其中 TE 是节点，张量是边。两个 TE 的输入和输出张量决定了它们之间的依赖关系。

4.5.3　全局计算图分析

SOUFFLE 支持 TE 依赖图上进行两种类型的全局分析：第一种是寻找张量数据复用机会，第二种是抽取出 TE 表达式中元素级别的数据依赖关系。

4.5.3.1　寻找数据复用机会

通常张量会在时间维度和空间维度被复用，与经典的缓存优化类似，SOUFFLE 可以把张量缓存在 GPU 片上存储空间，避免代价高昂的全局内存读写。如

图 4-1中所示，最开始的三个 *GEMM* 的算子共享同一个输入的张量，我们称这个张量在空间上被复用了。通过把这三个 *GEMM* 融合为一个 kernel，就可以只加载一次这三个 *GEMM* 共享的输入张量。这种类型的复用机会在 DNN 中很常见，在循环神经网络（Recurrent Neural Network, RNN）[196]，卷积神经网络（Convolution Neural Networks，CNN）[4,188,197] 和基于 Transformer 的模型等[9,198]DNN 模型中都存在类似的结构。第二种数据复用的机会可以被称为在时间维度的复用。仍然以图 4-1中的 BERT 模型为例，算子 *arithmetic operator 0*（标记为 *A0*）的输出被算子 *reduction operator 1* 和算子 *arithmetic operator 2* 访问。如果把把时间上先后被访问的张量缓存到片上内存，同样可以避免多次的内存访问，实现数据复用。

Souffle 从整个计算图上来寻找数据复用的机会。首先，Souffle 遍历 TE 程序，从中寻找出所有被访问了多次（一次以上）的张量。将张量与访问了它的算子记录为一个词典（Map）：$s(t_i) = \{op_j, ..., op_k\}$。这个词典的值（value）中所有的算子 op_k 都访问了同一个张量 t_i。

4.5.4 TE 内的元素级别数据依赖关系分析

Souffle 也可以捕捉到 TE 的输入和输出张量之间元素级别的数据依赖关系。TE 中的输出张量的形状决定了循环的迭代空间（Iteration Space），迭代变量和规约变量共同描述了输入张量的数据空间（Data Space）。本文发现，**与其构建复杂的从输入张量到输出张量的数据映射关系，不如考虑每个输出元素依赖哪个（或哪些）输入张量元素**。利用数据流图静态单赋值的特性，构建从输出到输入元素的依赖关系更加符合 DNN 计算流图的本质。而且这个信息足够在 TE 调度变换的阶段进行规约算子融合。而 TensorRT 很难从算子中抽取出详细的元素级别的依赖信息，因而难以支持规约算子的融合。

本文核心观察是 TE 内部的元素之间的依赖关系可以被分为两个类别。首先，对于没有规约维度（详见 4.3章）的 TE，所有的输入张量都是只读的（Read-Only），而所有的输出张量都是只写的（Write-Only）。对于这种类型的 TE ，每个输出张量都只依赖于一个输入张量的元素，称为“一依赖一”（*One-Relies-on-One*）。第二，对于有规约维度的 TE，每个输出张量的元素依赖于所有的在规约维度上的张量，称为“一依赖多”（*One-Relies-on-Many*）。有了以上的观察，和基于算子的依赖分析与基于源代码的依赖分析相比，本文提出的方法能够极大地简化依赖分析的过程。

考虑如图 4-4所示的 TE 中元素的依赖关系。*A_exp[i, j]* 只依赖于 *A[i, j]*，这就意味着只要 *A[i, j]* 被计算出来，*A_exp[i, j]* 的计算就可以立即开始。作为对比，*A_sum[i]* 依赖于 *A_exp[i, rk]* $\forall rk \in [0, 768)$，也就是说每个 *A_sum[i]* 依赖所有的 *A_exp[i, rk]* 中 *rk* 从 0 到 768 的元素。只有等这 768 个元素都计算完成，才能最终算出 *A_sum[i]* 的值。张量 *A_sum* 和 *A_exp* 之间就是 *one-relies-on-many* 的关系。

***One-Relies-on-One* TE 的变换。** Souffle 目前采用基于柯西仿射（或称为“准仿

射”，quasi-affine）[199,200] 从输出张量到输入张量角标映射的方法来表达 TE 内部元素之间的依赖关系。对于 *One-Relies-on-One* 类型的 TE，从输出张量的一个元素到输入张量的一个元素的依赖关系可以被表示为如下的形式：$M\vec{v}+\vec{c}$。其中，$\vec{v}$ 代表输出张量的角标，M 是一个 $\mathbb{Z}^{n\times m}$ 空间中的常数矩阵，$\vec{c}$ 是 $\mathbb{Z}^n$ 空间中的常数向量。在这里，m 是输出张量的维度，n 是对应的输入张量的维度。注意，多个输出张量的元素可能会依赖同一个输入张量的元素。例如，在图 4-4中，所有的 *A_softmax[i, j]*（$j \in [0, 768)$）都依赖于输入元素 *A_sum[i]*。从 *A_softmax[i, j]* 到 *A_sum[i]* 的映射的矩阵可以被表示为：

$$\begin{bmatrix}1 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0 & 0\end{bmatrix}, 0\le i<384, 0\le j<768$$

***One-Relies-on-many* TE 。**对于一个 *One-relies-many* 类型的 TE, Souffle 根据迭代空间和输入张量的角标函数来抽取出每个输出张量元素所依赖的输入张量元素的集合。因为规约变量所在的维度的长度是一个常量，依赖关系可以被表示为如下的形式：$\vec{v}\rightarrow f(\vec{v},\vec{r}), \forall\vec{r}\in\mathfrak{R}$。其中，$f(\vec{v},\vec{r})$ 是从输出角标 $\vec{v}$ 到输入张量的角标的映射函数。$\vec{r}$ 中保留了所有的规约变量，$\mathfrak{R}$ 是所有规约变量以及其对应的迭代区间（Iteration Domain）。例如，如图 4-4中所示，第二个 TE 的依赖关系可以被表示为：$(i)\rightarrow\{(i,j)|\forall 0\le j<768\}, 0\le i<384$。

4.5.4.1　TE 的特征刻画

Souffle 通过分析每个输出元素所需要的计算量来把 TE 划分为访存密集或者计算密集，这部分信息会发送给章节 4.5.7的 TE 调度的部分。具体而言，Souffle 使用 $\prod_r^{\mathfrak{R}} domain(r)$ 来计算 TE 的计算复杂度，其中 r 是 TE 的规约变量，$\mathfrak{R}$ 是规约变量及其对应的迭代区间。对于没有规约变量的 TE，Souffle 将它们分为两种类型：对于简单的算术运算的没有规约变量的 TE，标记其计算复杂度为 $O(1)$。对于有复杂算术运算的（例如 *sin*），Souffle 标记其复杂度是 $O(10)$，表明其计算需要多个时钟周期才能完成。获得计算复杂度的估计后，Souffle 推测其计算访存比，以此来把 TE 划分为访存密集型（例如 *reduce_sum*）和计算密集型（例如 *GEMM*）。划分的阈值根据经验设置为 2，低于这个阈值表明这个 TE 是访存密集型的。

4.5.5　TE 程序切分

在 Souffle 中，首先需要将 TE 程序的切分为由 TE 组成的子程序，之后高层级 TE 变换、中间层的调度优化和后端的代码生成都在每个子程序上进行。一个 TE 的子程序可能包含多个算子，这些算子最终会被映射生成为一个 GPU 的 kernel。Souffle 综合考虑 DNN 计算图的全局分析结果、张量的信息以及 CUDA 的全局同步原语带来的限制来进行 TE 程序的切分。

4.5.5.1 TE 张量的间接访问

在 DNN 中中通常会有张量的间接访问，其访问输入张量的角标在编译时不能确定。例如 $C[i] = A[B[i]]$，其中 A 和 B 为输入的张量（只有在运行时才知道具体值），C 为输出的张量。编译器无法在不知道访问角标的前提下安全地进行 TE 的变化。对于这种情况，Souffle 在这个 TE 之前把 TE 程序划分为两个 TE 的子程序，之后单独地优化两个子程序，从而保证 DNN 模型编译的正确性。

4.5.5.2 全局同步原语

当进行指令调度的时候，Souffle 可能需要插入全局同步原语来在一个 kernel 的所有线程之间进行同步。具体的例子如图 4-1(d) 所示。这保证了在进行下一个算子（或 TE ）的计算之前，所有的线程都已经生成了其负责计算的结果。全局同步原语是通过 CUDA 提供的 cooperative group API [201] 来实现的。而使用这个 API 要求 kernel 的并行的维度和 kernel 使用的寄存器的数量（和 shared memory）的数量不能超过每个波面（Wave）所能并行执行的最大线程块数量。假设一个 GPU 的流处理器（Streaming Multiprocessor,SM）有 65536 个寄存器，一个 SM 上需要运行 4 个 block，每个 block 中有 256 个线程，每个线程占 64 个寄存器则同时运行 4 个 block 需要 $4\times256\times64 \rightarrow 65536$ 个寄存器。此时刚好能够满足全局同步的要求。而如果每个线程占 65 个寄存器，总共需要 $4\times256\times64 \rightarrow 65560$ 个寄存器，超过了 SM 的总寄存器数量，就不能够使用全局同步的原语。

Souffle 通过编译分析的方式来检查一个算子是否满足全局同步的约束。Souffle 首先将抽取 TE 程序中所有的计算密集型的 TE ，之后编译经过调度的 kernel，使用 CUDA 的 *cuobjdump* 工具从编译生成的二进制可执行文件中获取（1）cuda kernel 的 grid dimension 和 block dimension；（2）寄存器和共享内存的占用。因为调度和编译一个 TE 只需要进行一次，且相同的 TE 经常在模型中出现多次，所以判断 DNN 模型中计算密集型算子能否满足全局同步约束的开销是可控的。

Souffle 基于一个分析模型来确保 TE 程序的切分能够满足使用全局同步原语的要求。假设 GPU 总的寄存器（或共享内存）的数量是 C，一个 TE 子程序中 TE 的最大的 kernel 启动的维度是 max_grid，最大的寄存器/共享内存的占用是 max_acc。当要使用全局同步原语的时候，必须满足 $max\_grid \times max\_acc < C$。基于这个约束，Souffle 从一个 TE 开始，贪心地构造 TE 子程序。Souffle 从 TE 程序的输出张量开始，贪心的把产生当前输出张量的 TE 的前驱 TE 加入到 TE 子程序中。如果前驱 TE 能够满足约束，就将前驱 TE 加入子程序集合中；否则就创建一个新的 TE 子程序，并把这个前驱 TE 加入到其中。重复以上的过程直到程序中所有的 TE 都被加入到了某个 TE 子程序中。当然，Souffle 也有可能将 TE 子程序进一步切分成更小的子程序，以满足全局同步的约束。

```
# shape A1:(4,8),B1:(8, 16),A2:(2, 8),B2:(8, 16)
rk = te.reduce_axis((0, 8), name="rk")
C1 = te.compute((4,16), lambda i,j:te.sum(A1[i,rk]*B1[rk,j],axis=[rk]))
C2 = te.compute((2,16), lambda i,j:te.sum(A2[i,rk]*B2[rk,j],axis=[rk]))
```

Horizontal transformation

```
C = te.compute((4+2, 16), lambda i, j:
  te.sum(tir.if_then_else(i<4, A1[i, rk], A2[i, rk]) *
    tir.if_then_else(i<4, B1[rk, j], B2[rk, j]), axis=[rk]))
```

图 4-5 两个 *GEMM* 算子的水平变换

Figure 4-5 Horizontal transformation for two *GEMM*

4.5.6　TE 变换

如章节 4.5.3中所述，Souffle 获取到张量的复用信息和依赖信息之后，就开始寻求自动 TE 变换的机会来提高 TE 子程序的性能。Souffle 支持水平方向（Horizontal transformation）和垂直方向（Vertical transformation）的 TE 变换。其中，水平方向的变换主要应用在独立的 TE （相互之间没有数据依赖关系），而垂直方向的变换主要应用于多个连续的 TE （有直接的数据依赖关系且元素间的依赖关系是 *One-Relies-on-One*）。

TE 的结构等价。水平变换是基于两个 TE 的结构等价。在 Souffle 中，如果它们满足如下的约束条件，则可以认为是结构等价：（1）它们的迭代区间 $\mathfrak{R}$ 相等；（2）它们的运算符相等（例如，*tir.sum*，*tir.add* 等）；（3）它们的输出的张量的形状相同；（4）它们对应张量的形状和规约变量相同。

4.5.6.1　独立 TE 的水平变换

Souffle 会尽可能的将独立的 TE 合并为一个 TE 来增加并行度，以充分利用 GPU 的多个计算单元（通常有数百到上万个 CUDA 核）。对于包含规约计算的 TE ，Souffle 首先检查两个 TE 是否是结构等价的。如果是，则 Souffle 将他们的输入张量进行连接操作（Concate），然后在迭代空间增加一个维度，等价于将这几个 TE 合并加到了一个循环中，循环的长度是这几个 TE 的数量。如果这几个 TE 的输出只有一个维度不同（即输出张量可以被连接到一个张量），满足结构等价的其他条件，Souffle 基于连接的区间大小增加谓词（条件判断）来选择对应的输入张量，最后重写 TE 。如图 4-5所示为两个 GEMM 水平合并的例子。两个 TE 都共享通一个规约变量 *rk*。第一个和第二个 TE 的输出张量的形状分别是 $(4, 16)$ 和 $(2, 16)$。两个 TE 的输出张量可以在第一个维度是被连接到一起，形状为 $(4, 16)$。这个方法和函数合并类似[202]。

4.5.6.2　面向 *One-Relies-on-One* TE 的垂直变换

Souffle 对输出张量对输入张量是 *One-Relies-on-One* 类型的 TE 进行垂直的变换以减少 TE 语句的数量。目前通过转换和连接 TE 中输出和输入张量的角标

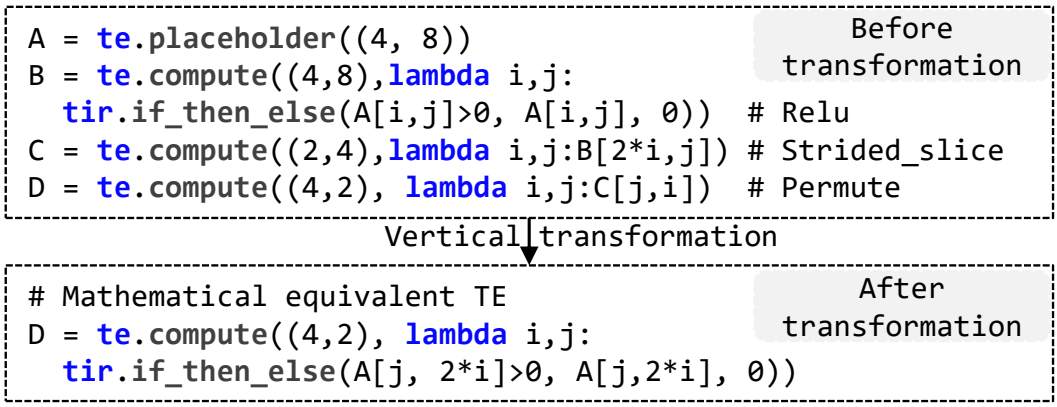


图 4-6 垂直 TE 变换的例子

Figure 4-6 Horizontal transformation for two *GEMM*.

映射函数来实现减少内存访问。为了达到这个目的，Souffle 首先从子 TE 到父 TE 把所有的角标映射函数连续地乘起来，最终实现把连续的多个 TE 的映射函数转换为数学上等价的映射函数。

假设有 n 个 TE,$te_0, te_1, \cdots, te_i, \cdots, te_{n-1}, te_i$ 的输出是 te_{i+1} 的输入。te_i 的输出和输入直接的角标映射函数可以被表示为 $f_i(\vec{v_i})$。（注意，这里只列举了输出张量对一个输入张量的映射，对于多个输入张量的 TE 可以同样的适用。）从 te_{i+1} 到 te_i 的张量计算函数可以用如下的表达式计算：

$$f_{i+1,i}(\vec{v_i}) = f_{i+1}(f_i(\vec{v_i})) = M_{i+1} \times (M_i + \vec{c_i}) + \vec{c_{i+1}} \tag{4-1}$$

以图 4-6为例，三个 TE 的映射函数 A, B 和 C 可以被变换为一个数学等价的函数，从而可以将 TE 的数量从三个减少为一个：

$$\begin{bmatrix}0 & 1\\ 1 & 0\end{bmatrix}\times\left(\begin{bmatrix}2 & 0\\ 0 & 1\end{bmatrix}\times\left(\begin{bmatrix}1 & 0\\ 0 & 1\end{bmatrix}\times\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}\right)+\begin{bmatrix}0\\ 0\end{bmatrix}\right)+\begin{bmatrix}0\\ 0\end{bmatrix}\Rightarrow\begin{bmatrix}0 & 1\\ 2 & 0\end{bmatrix}\begin{bmatrix}i\\ j\end{bmatrix}+\begin{bmatrix}0\\ 0\end{bmatrix}$$

基于以上的方法，Souffle 迭代地把多个 *One-Relies-on-One* 的 TE 变换为一个 TE ，直到没有可变换的 TE 。之后，如下一章节中描述的，Souffle 会对 *One-Relies-on-Many* 的的 TE 进行调度的优化。与 TensorRT、Apollo 和 Ansor 手工制定的规则相比，基于数学上等价变换的方法具有更好的普适性，能够适应各种不同类型的算子而无需手工制定规则。

4.5.7 面向 *One-Relies-on-Many* TE 的调度融合

在进行完 *One-Relies-on-One*TE 的合并之后进行 *One-Relies-on-Many* 的 TE 的变换。在章节 4.5.4.1中，本文将 TE 划分为了访存密集型和计算密集型。对于访存密集型的 TE （可能为包含规约操作的 TE ），Souffle 寻求能够复用计算结果的机会以减少代价高昂的内存访问。而对于计算密集型的 TE ，Souffle 则会利用 Ansor 等来生成能够达到最佳性能的调度。

Souffle 使用了一个简单但却有效的启发式策略来为 TE 子程序中的 TE 生成调度。Souffle 从 TE 子程序的输出张量开始，从子 TE 到父 TE 进行扫描。对于一个访存密集型的规约计算的 TE r，Souffle 查看产生 r 的输入张量的 TE 。如果 r 的后继的 TE 是一个 *One-Relies-on-One* 的 TE，那么规约计算的 TE 的调度就使用后继的 *One-Relies-on-One* 的调度，这样就能够在两个 TE 之间复用张量、同时还能够提升并行度。而对于计算密集型的 TE ，Souffle 使用调度优化器（例如 Ansor）为其生成调度，并把它的调度传播到所有其他非计算密集型的(例如访存密集型）TE 。这样既能够保证计算密集型 TE 的性能（直接使用调度优化器生成调度以最大化性能)，又能够在片上内存复用访存密集型 TE 产生的张量（访存密集型 TE 使用计算密集型的调度)。

4.5.8　跨算子联合优化

Souffle 的一个独特的优势就是能够跨算子边界调度计算原语以进行指令级调度。具体而言，Souffle 能够跨算子边界的重排指令以提升指令级并行性。例如，NVIDIA A100 GPU 能够并行地发射和执行访存指令(例如 *LDGSTS.E.BYPASS.128*，异步的从全局内存拷贝 128 比特数据到共享内存)和算术运算指令(例如 *HMMA.16818.F16*，在张量核心 Tensor core 上执行 $m16n8k16$ 的矩阵乘)。Souffle 实现了一个面向共享内存简单的软件管理的缓存，以在不同的 TE 之间复用已经加载到共享内存中的张量。

Souffle 在为 TE 子程序生成调度之后进行联合的优化。首先 Souffle 把子程序中单独的 TE 合并为一个联合的调度。基于全局的分析，Souffle 可以推断出 TE 之间的张量是否有数据依赖关系。之后，Souffle 寻找把访存指令与计算指令并行执行的机会。对于没有数据依赖关系的访存指令与计算指令，Souffle 首先把访存指令向前移动到计算指令之前，之后用异步（*asynchronous*）访存指令代替原来的同步访存指令，最后在访存指令原来的位置插入对应的内存屏障指令，保证数据在被使用之前已经被正确的加载到共享内存中。这样，Souffle 就能够实现如图 4-1(d) 中所示的，把加载张量的操作和计算的操作重叠执行。

Souffle 可以自动地计算一个 kernel 所需要的共享内存数量。通过线性扫描一个 TE 子程序的调度，在编译的时候进行指令调度，直到共享内存不够使用。这时候 Souffle 会释放掉最早分配的共享内存，并在访问到要释放的共享内存的指令之前加入访存的指令，加载回因为共享内存不足而释放掉的数据。由于有了全局的调度，Souffle 可以尽可能的在 TE 之间复用张量，在共享内存空间足够的情况下，让张量尽可能的保持在共享内存中。Souffle 的缓存管理策略是简单的先入先出（First-In-First-Out，FIFO)，张量的活跃区间的信息可以从章节 4.5.3中的分析得到。

本章工作的亮点：不像以前的算子融合的工作（例如 Apollo 等)，Souffle 采取了不同的方法来跨算子的边界进行指令调度和复用片上缓存的数据，从而可以达到更好的性能。如果没有全局分析，就不能够完成联合优化，这也是本章

工作的一个核心的亮点。

4.6 实验设置

实验平台。本章的实验平台是一个 GPU 服务器，包括一个双路 20 核心、2.50GHz 的英特尔 Xeon Gold 6248 CPU，装备有 768GB 的 DDR4 内存和一个 40GB 显存的 PCIe 版 NVIDIA A100 GPU。服务器运行 Linux 内核版本为 5.4.55 的 Ubuntu 18.04.5 系统。本实验使用 CUDA 11.7，并开启“-O3”的编译优化选项。

DNN 评测负载。

本实验在有代表性和多样性的 DNN 工作负载上衡量 SOUFFLE 的性能，包括自然语言模型（BERT，[10]），计算机视觉模型（Swin-Transformer[198]，简写为 Swin-trans），知识发现的模型（MMoE[203]）。也包括经典的卷积神经网络，如 ResNext[197] 和目前性能最优的卷积神经网络 EfficientNet[204]，以及循环神经网络 LSTM[6]。DNN 模型的具体配置见表 4-2。

4.6.1 有竞争力的对比对象

本实验将 SOUFFLE 与四个当前最优的算子融合的工作做对比：（1）Ansor。这是目前最好的基于 TVM 的 DNN 算子优化器。它基于自动调优（Auto-tuning）的技术来为 TE 找到好的调度。它也包含一个简单的融合的机制，可以将多个 TE 合并生成为一个 kernel。本实验使用 TVM 0.8 版本中的 Ansor。**（2）TensorRT。**它是 NVIDIA 专门为 NVIDIA GPU 优化的 DNN 模型推理框架。本实验使用的是 TensorRT 7.2.0 版本。**（3）Rammer。**这是工业界知名的 DNN 编译器，也被称为 NNFusion[62]。它在编译的过程中将算子水平合并到不同的线程块中，以并行地执行多个算子。它也探索了算子间和算子内的协同调度，本实验使用最新版本的 NNFusion。**（4）Apollo。**它代表了目前最先进的为了推理优化的算子融合的框架[121]。Apollo 不仅考虑访存密集型算子，也考虑计算密集型算子的融合。它使用手工的规则进行算子融合，并且也优化了独立的算子之间的并行执行。

4.6.2 性能报告

为了测量 DNN 模型的性能，本实验首先在实验平台上跑 1000 次使机器“活跃起来”，然后在相同的输入上重复跑 DNN 模型 10000 次。重复这个过程三次并报告运行模型的端到端的执行时间。试验发现每次运行的误差值非常小，通常在 2% 以内，所以只报告平均时间。

表 4-2 DNN 基准测试集

Table 4-2 DNN models and datasets

模型	数据集	应用领域	模型参数
ResNet	ImageNet	计算机视觉	#layers:101, bottleneck width: 64d
EfficientNet	ImageNet	计算机视觉	使用原始论文中的 Efficient-b0 的配置
Swin-Transformer	ImageNet	计算机视觉	base 版本, 其中 patch= 4，window size = 7
BERT	SQuAD	自然语言模型	BERT-base，layers=12，# of head = 12
LSTM	synthetic	自然语言模型	input length: 100, # hidden size: 256, layer: 10
MMoE	synthetic	知识发现	使用论文中 MMoE 的 synthetic 版本[203]

4.7　实验结果

4.7.1　总体性能

表 4-3展示了六个模型在五个测试程序上的端到端的执行时间。注意有些编译器不能编译某些模型（例如 Apollo 编译 LSTM 超时、Rammer 编译 Efficientnet 遇到了不支持的算子而失败)。表 4-3表明，Souffle 的在所有的 DNN 模型上的性能都显著的优于基准测试程序。相比 TVM 的 Ansor，Souffle 能够达到平均 3.94×（最高的 8.47×）的加速比；相比 NVIDIA 专门调优的 TensorRT，Souffle 能够达到平均 3.11×（最高 5.17×）的加速比；相比 Rammer 和 Apollo 也能够达到相似的性能提升。Souffle 通过全局的分析（包括张量级别的分析和张量中元素级别的分析）能够发现更多的优化机会，这些全局的分析最终能够使得 TE 变换和联合优化成为可能，并最终达到更高的性能。

4.7.2　性能刻画

Kernel 调用数量的减少。 Souffle 的图划分策略旨在在 GPU 共享内存允许的情况下，创造大的 TE 子图（从而生成更少的 kernel)。这个策略减少了 kernel 调

表 4-3 端到端的模型执行时间（单位：*ms*，越低越好）

Table 4-3 End-to-end model runtime (*ms*) - lower is better

模型	Ansor	TensorRT	Rammer	Apollo	Souffle
BERT	6.13	2.27	2.19	3.29	**1.32**
ResNeXt	20.50	22.92	11.69	22.80	4.43
LSTM	6.78	3.42	1.72	Failed	**0.80**
EfficientNet	0.94	1.32	Failed	2.3	**0.66**
Swin-Trans.	5.81	3.80	Failed	10.78	**1.78**
MMoE	0.023	0.083	Failed	0.049	**0.019**

表 4-4 Kernel 调用数量的减少

Table 4-4 Kernel calls reductions.

模型	TensorRT	Apollo	Souffle
BERT	312	240	24
ResNeXt	2406	1226	105
LSTM	662	Failed	1
EfficientNet	187	273	66
Swin-Trans.	637	1014	53
MMoE	20	10	1

表 4-5 全局内存访问（M bytes）的降低

Table 4-5 Device Memory Transaction (M bytes) Decreme

模型	TensorRT	Apollo	Souffle
BERT	527.52	880.54	226.8
ResNext	622.17	436.14	470.16
LSTM	126.84	Failed	10.6
EfficientNet	96.37	127.36	86.58
Swin-trans.	397.82	1309.03	282.9
MMoE	0.061	0.063	0.058

用的数量，以及对应的 kernel 启动的开销，以及读写全局内存的开销。例如，在 BERT 模型中，TensorRT 和 Apollo 分别生成了 312 和 240 个 kernel。作为对比，Souffle 将 kernel 调用的数量减少到了 24 （约 10× 的 kernel 数量的减少)。

全局内存访问 (*M* bytes) 的降低。 Souffle 通过 TE 变换减少不必要的片外内存访问，以及通过联合优化隐藏部分数据加载的延迟来减少 DNN 端到端的执行时间。由于张量都尽可能的被缓存到了共享内存上，并且张量在时间和空间维度被尽可能地被复用，因而内存访问量极大地降低。例如，在 BERT 中，TensorRT 和 Apollo 分别从全局内存加载了 527.5MB 和 880.5MB 的数据。作为对比，Souffle 减少内存数据的加载量到 226.8MB。

4.7.2.1　BERT 的性能分解研究

实验中一个一个的打开 Souffle 的优化选项，以验证 Souffle 的每个优化的有效性。本实验的基准是 TensorRT。首先打开 TE 的水平和垂直的变换（章节 4.5.6.1和章节 4.5.6.2)，之后打开全局优化以利用全局同步原语 (章节 4.5.5.2)，最后打开联合优化（章节 4.5.8)。

TE 的*水平和垂直变换*将计算的延迟从 189μs 降低到了 136μs。*全局优化*的技术通过使用全局同步原语将张量计算的中间结果缓存到共享内存，又减少了 10μs 的延迟。之后的联合优化通过让计算和访存指令重叠并行执行，又减少了 16μs。

本文把 BERT 的 kernel 分为计算密集的 kernel(例如 GEMM)和访存密集型的 kernel（例如 Softmax）来分析 Souffle 性能收益的来源。使用 nsight-compute[205] 来搜集 TensorRT 的 kernel 的延迟。由于多个计算密集和访存密集型的 kernel 被 Souffle 合并到了一个 kernel，本文使用 CUDA 提供的 *clock64* 的 API 在 CUDA kernel 中代码段中插桩，然后求两个之间的插值以作为代码段的延迟。最终，在 BERT 的一个层中，TensorRT 生成了 26 个 kernel，而 Souffle 只产生了 2 个 kernel。内存密集型的延迟从 101.7μs 降低到了 25.5μs，这是因为 Souffle 把所有的中间结果都缓存到了片上的共享内存中。

4.7.2.2　LSTM 的性能分解研究

在本节以 LSTM 为例，来说明 Souffle 所发掘的新的优化机会。对于 LSTM，Souffle 能够达到非常显著的加速效果，相比 TensorRT 可以达到 4.27×，相比 Rammer 可以达到 2.15× 的加速比。Rammer 是对比框架中 LSTM 性能最好的，因此本文将 Souffle 和 Rammer 做详细的对比分析。图 4-7展示了 Rammer 和 Souffle 各自的融合策略。本实验中衡量的 LSTM 模型包括 10 层（在图 4-7中垂直方向排列)，每一层都有其自己的权值张量（图 4-7中标记为 W 和 U)，隐藏状态 h 和输出 c。在每个时间步，第 n 层使用其权值张量 W_n 和 U_n、隐藏状态 h_n 和 $(n-1)$ 层的输出 c_{n-1} 进行通用矩阵向量乘操作（简写为 *gemv*)，*gemv* 产生的输出更新其隐藏状态 h_n 和为当前的时间步生成输出 c_n。图 4-7在水平方向将

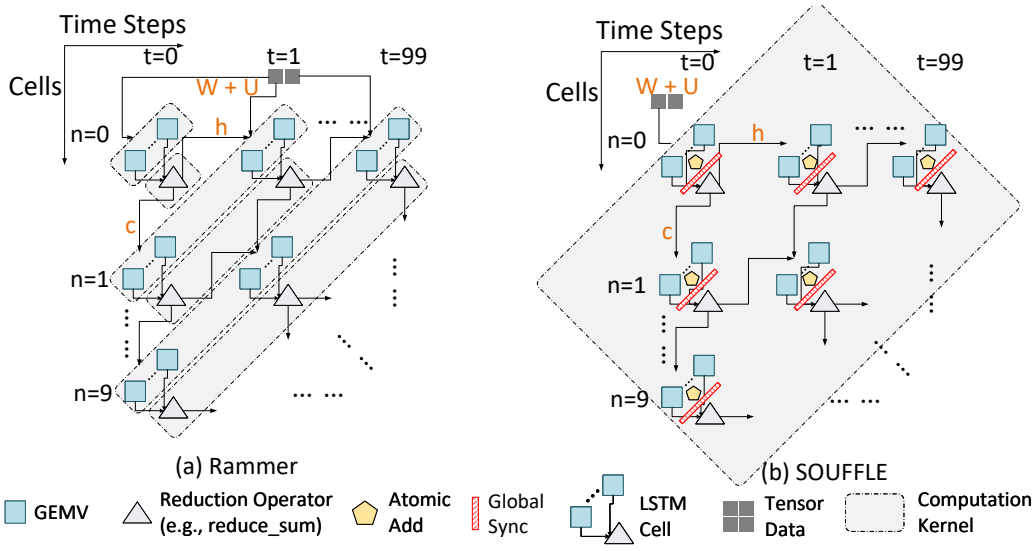


图 4-7 Rammer(a) 和 Souffle (b) 将 LSTM 映射到 CUDA 计算核的方式

Figure 4-7 How Rammer (a) and Souffle (b) map a LSTM graph into kernels

表 4-6 LSTM 所有计算核性能计数器的结果

Table 4-6 Total performance counters of all kernels in LSTM

衡量指标	Rammer	Souffle
从全局内存访问的字节数	1911.0MB	21.11MB
访存单元（Load Store Unit）流水线利用率 (LSU)	20.2%	35.4%
乘加单元（Fused Multiply Add Unit）流水线利用率	8.0%	19.0%

时间步都展开了。

从图中可以看出，在对角线方向的算子是独立的，即它们之间没有数据依赖。Souffle 和 Rammer 都能够发现这样的优化机会，即将波面上的算子都合并到一个计算核的不同线程块上。

然而，由于 Souffle 可以进行章节 4.5.3中描述的全局分析，Souffle 发现 LSTM 每层中的权值张量 W 和 U 在不同的时间步是被复用的，因此可以考虑将不同时间步的算子进行融合，从而发掘复用权值张量的机会。最终，通过在代码中插入 *global synchronization* 指令，Souffle 能够为 LSTM 模型只生成一个计算核。直观的，Rammer 版本的 LSTM 需要在每个波面（对角线上的算子合并为了一个波面）上都加载一次权值，而 Souffle 只需要加载一次张量 W 和 U 到 GPU 的共享内存中，即可在不同的时间步复用，从而极大地减少了访存的开销。

指令分析。表 4-6展示了 Rammer 和 Souffle 的 LSTM 模型 GPU 的性能计数器的结果。指标主要分为两个方面：

- **减少访存。**表 4-6展示了 Souffle 可以将 LSTM 模型的 GPU 访存数量从 1911MB 减少到 21.11MB。这是因为通过精确的全局张量分析，Souffle 将所有

的权值都缓存在了共享内存中，从而极大地减少了访存。

- **增加流水线利用率。**表 4-6表明 SOUFFLE 可以同时增加访存单元（Load store unit, LSU）和乘加单元（Fused multiply add unit，FMA）的利用率。这是因为 LSU 单元可以从快速的片上共享内存取数，避免等待代价高昂的全局内存的访问，因而 FMA 单元的等待时间也大大降低。

4.7.3　优化开销

SOUFFLE 使用 Ansor 和 TVM 来搜索和生成 TE 的调度。SOUFFLE 的开销主要来源于两层的依赖分析、模型切分、TE 调度搜索以及全局优化。本实验测量了 Ansor 和 SOUFFLE 从加载 DNN 模型开始到生成 CUDA 程序的时间。在 6 个负载上的实验结果表明 SOUFFLE 在 Ansor 的基础上增加了 63 秒的时间开销，考虑的模型只需要被生成一次就可以被长期的部署，这个开销是可以忽略不计的。最主要的时间开销主要来自于 Ansor 进行 TE 调度搜索的时间，对于 Swin-Transformer 来说，Ansor 需要几个小时才能完成搜索。这个开销可以通过使用更好的调度搜索算法来降低，（例如可以使用 Roller[65]）。SOUFFLE 的工作和 Ansor 以及 Roller 的工作是正交的。

4.8　本章小结

本章提出了 SOUFFLE 以提升 DNN 模型在 GPU 上的推理性能。SOUFFLE 通过在计算图上进行全局分析跨算子地发掘优化机会；通过把张量表达式合并为一个子程序，最终生成一个计算核。SOUFFLE 在子程序上进行局部的优化，包括保持语义不变的张量表达式变换、指令调度以及数据复用。与当前最优的四个 DNN 框架对比，SOUFFLE 能够发掘更多的优化机会，达到更好的性能。

当然，SOUFFLE 的工作也存在的提升的空间和未来可以继续研究的工作。

多 GPU 支持。SOUFFLE 目前只支持在一个 GPU 上进行推理。一些大模型会被裁剪为小的模型，从而能够在一个 GPU 上完成推理[206]。SOUFFLE 也可以被集成模型并行的框架中以支持多 GPU 的推理[207]。

TE 程序自动划分的代价模型。 SOUFFLE 将 TE 编译为二进制，然后从中抽取出占用的共享内存数量等信息。未来通过构建代价模型的方式，从 TE 中直接获取这样的信息[208]。

更多联合优化的机会。目前本章联合优化主要集中在探索指令集并行。未来希望能够将联合优化应用在更多跨计算核的场景，例如常量折叠和与张量相关的转换[209] 等。

第 5 章　DNN 推理数据前后处理的编程框架

5.1　相关背景与研究动机

在第 3和 4章中刻画了 DNN 模型在多种软硬件配置下的性能表现，并分别在编程框架和编译器中，针对计算图做了自动调优和代码生成。然而，深度学习应用不止包含 DNN 模型，还包括很多非 AI 的程序代码。本章主要针对深度学习应用中的非 AI 代码在智能处理器上的部署和性能优化进行研究和讨论。

在目前采用了深度学习技术的应用中，DNN 模型虽然占据了核心地位，但是在运行 DNN 推理之前通常需要对数据进行一些预处理（Pre-processing），执行完成之后需要对 DNN 模型产生的数据做后处理（Post-processing）。例如，在谷歌相机图像优化的流水线中[210]，需要先对相机拍摄的图像进行下采样（预处理），之后运行一个 DNN 模型对图像进行语义分割（DNN 模型推理），之后进行上采样、色调映射、降噪等（后处理）。类似的流程还包括对图像的自动高动态对比度（High Dynamic Range, HDR）[211,212]、自动驾驶中图像预处理[213,214]、推荐系统中的聚类[215] 和相似度度量[216] 等。DNN 推理和数据的前后处理紧密的结合在一起，共同组成了应用中对数据处理的流水线。因而，对数据的前后处理也是影响应用端到端延迟的重要因素。

各种面向深度学习的神经处理芯片（Neural Processing Unit，NPU，为简便后文中将这类芯片统称为 NPU），例如寒武纪的 MLU 系列[24,217,218]，谷歌的 Tensor Processing Unit（TPU）[25]、GraphCore、华为昇腾系列[58] 等被开发出来并且已经在商业系统中得到了广泛的使用。这些 ASIC 芯片 DNN 的推理性能相比 CPU 有个一至两个数量级的提升。由于 DNN 主要计算量来自于卷积和矩阵乘运算，总的来说，这些芯片通常都包括多个加速矩阵乘的计算单元，例如谷歌 TPU 中的脉动阵列、华为昇腾中的矩阵立方（Matrix Cube）。此外，这些 ASIC 芯片还包括一些向量部件用于加速一些访存密集型的运算，以及一些标量单元以细粒度的操作张量元素。

NPU 极大地加速了 DNN 的推理，而深度学习应用中处理 DNN 的输入和输出数据又对 CPU 的处理能力带来了极大的挑战。研究人员也发现，由于 DNN 模型推理越来越快在当前数据中心 DNN 推理中，CPU 对图像的预处理已经成为系统的瓶颈[67]。互联网中超过 80% 的流量都是视频流量[219]。海量的图片和视频为 CPU 的处理带来了巨大的挑战。以一张全高清分辨率为 1920×1080 的 JPEG 图片为例，在 Intel(R) Xeon(R) Gold 6151 CPU 上，使用 OpenCV-4.6 版本，单个核心解码的耗时为 21.20 毫秒，将解码完成的图像传输到华为昇腾 310 推理卡（简称为 Ascend310）上需要 1.06 毫秒，总耗时为 22.26 毫秒。而如果先将 JPEG 图片传输到 Ascend310 上（耗时 0.34 毫秒），再调用 Ascend310 解码（耗时 4.37 毫秒），则总耗时为 4.71 毫秒。相比 CPU 解码可以获得 4.73× 的加速比。而在 Ascend310 上进行一次 ResNet-50 的推理也只需要 3.42 毫秒。因而，将原始的图

像直接传输到 NPU 上进行处理，不仅能够减少应用的端到端延迟，又能够降低对 CPU 的资源占用，达到一举两得的效果。

然而，将 DNN 的预处理和后处理迁移到 NPU 上运行，存在着以下三个方面的挑战：

- **编程难。**各家提供的 NPU 芯片的编程接口差异巨大。与 CPU 的单指令流单数据流和 GPU 的单指令流多线程（Single Instruction Multiple Thread, SIMT）编程模型不同，NPU 芯片由于架构的差异其编程和 CPU 以及 GPU 有很大的差异。例如 Google 的云端 TPU 需要通过 TensorFlow 才能够调用；Edge TPU 需要配合 Edge TPU Compiler[220] 将 TFLite[221] 的模型编译后才可以部署在 Edge TPU 上；华为的 Ascend 系列需要通过 ACL（Ascend Computing Language）[222] 来调用和执行。由于主要面向 DNN 模型模型优化，大多数的 NPU 加速器没有面向开发者提供像 CUDA[60] 一样的细粒度对 NPU 进行编程的语言，而是提供了优化的 DNN 算子实现。然而，对数据的处理需要细粒度的操作数据元素，从而为数据的前后预处理带来挑战。
- **优化难。**第一，NPU 的计算单元通常采用更低位的比特数（即低精度部件，例如 8 位的整数、16 位的浮点数），这是因为 DNN 模型可以容忍一定的数值精度损失而保持推理结果仍然保持准确[223–225]。然而如果应用到通用的计算中需要对计算进行调整，否则很容易出现数值溢出导致结果错误。第二，通用数据处理和 DNN 应用有显著差异。NPU 的核心为计算矩阵乘的计算单元，而数据处理通常多为访存密集的计算。如何充分利用 NPU 的强大算力，以优化数据处理是一个难点。第三，NPU 加速器的性能特征差异很大，不同 NPU 加速器的内存带宽、片上缓存大小、向量单元长度、并行的核心数差异很大。这些隐藏的体系结构特征为优化数据处理程序带来了巨大的挑战。
- **部署难。**将应用部署到 NPU 上需要用户细粒度的调用专为 NPU 开发的 API，并且需要用户将数据处理程序与 AI 推理模型集成到一个应用中，使得难以部署到 NPU 上。如何为用户提供简易的接口使得数据处理程序与 AI 应用无缝的对接到一起是亟需解决的问题。

为了能够加速 AI 应用中的前后数据处理，使得 NPU 能够执行更加通用的计算任务，让应用程序端到端的运行在 NPU 上，本章提出了一个面向 NPU 的数据处理框架 GDPNPU（General Data Processing for Neural Processing Unit）。GDPNPU 针对以上的几个难点，提供了一系列简单的编程接口和库，帮助用户快速的在 NPU 上开发针对 DNN 推理数据处理程序。通过本文提出的 GDPNPU，用户能够将传统需要在 CPU 上处理的程序迁移到昇腾加速器上，利用昇腾加速器上性能强大的向量单元和矩阵单元进行处理。GDPNPU 的提出也证明了 NPU 不仅可以加速 DNN 模型的推理，同样能够进行多样的图像处理等更加通用的应用，为将来探究 NPU 加速器加速非 DNN 类应用做了一些探索性工作。

GDPNPU 允许用户基于 TensorFlow 和 PyTorch 来写数据预处理的程序，之后通过 GDPNPU 提供的编程接口即可直接编译运行在昇腾加速器上。GDPNPU 还

可以根据用户提供的 TE 表达式，生成细粒度的对张量相邻元素做运算的张量计算图代码。GDPNPU 封装了昇腾计算语言（Ascend Computing Language，ACL），避免用户操作繁琐的编程 API，而只需要基于熟悉的深度学习编程框架来写数据处理程序。GDPNPU 核心是自动数值精度转换和计算图层级的自动优化。GDPNPU 可以自动地根据输入数据的数值范围和算子的计算性质，将张量的数值进行缩放，因此可以充分利用昇腾加速器在较低精度下（例如 FP16）的性能，同时最小化精度降低对计算结果的影响。

由于昇腾加速器专为加速 DNN 模型设计，如果直接调用 DNN 的算子进行图像处理任务，性能会难以保证。本章针对昇腾加速器的向量指令和矩阵运算进行了基准测试，衡量其硬件的性能特征。基于性能测试的数据，可以得到其峰值的内存带宽、在给定规模下各计算单元可以达到的性能等参数。之后，针对数据处理中常见的运算，GDPNPU 可以进行自动算子替换以优化其推理性能。

本工作在华为云的弹性云服务器上实现了 GDPNPU 的系统，并实现多种典型的图像处理应用与数据预处理应用，包括边缘检测（Canny Edge Detection）、高斯模糊（Gaussian Blur）、色调映射和 K-近邻（K-means）等。针对系统的一系列实验表明 GDPNPU 能够实现传统在 CPU 和 GPU 上才能运行的图像处理应用可以运行在 NPU 上。，且相比于 CPU 在数据处理任务上最高可以实现 6.45× 的加速比。

本章的工作主要有以下的贡献：（1）本文提出了一个面向 NPU 的图像与数据处理的编程框架 GDPNPU，从可以在 NPU 上端到端的执行图像处理与数据处理任务，探索实现了在 NPU 上运行 OpenCV 图像处理任务的可行性与性能。（2）本文提出了自动精度调优，在 NPU 上使用低精度计算单元时对输入数据进行缩放，避免发生溢出，并提高推理性能。（3）本文针对图像与数据预处理程序在 NPU 上运行的特点，自动进行算子替换，相比原始的实现可以显著的加速数据处理的性能。

5.2　背景介绍

本章的工作主要基于华为的昇腾 310（Ascend310 AI 处理器，以下简称 Ascend310）来验证本章提出的编程框架。Ascend310 能够以 8 瓦特的功耗在 FP16 的精度下达到 11 TFLOPS 的峰值计算能力。之所以选择 Ascend310 有以下四方面的原因：（1）Ascend310 可以直接在华为云上购买得到，因而方便其他的研究人员进行继续开发和研究。（2）Ascend310 具有非常丰富的开发者文档和工具链，用户既可以选择调用已有的算子库，也可以自定义开发算子，而其他的绝大多数仅支持端到端的推理 DNN 模型。（3）Ascend310 提供了细粒度的获取算子执行时间、硬件资源利用率以及运行时库的执行时间等接口，方便分析具体的性能表现。（4）Ascend310 主要面向推理优化，其设计功耗只有 8 瓦特，与典型的英特尔 CPU 服务器的单核心功耗接近，具有很高的性能功耗比。

本节简要介绍 Ascend310 的硬件架构，更详细的内容可以参考 Ascend310 的编程书籍[59]。与 Google TPU 的架构不同，Ascend310 更像是一个 SoC。Ascend310 最核心的计算部件是 AI Core，每个 Ascend310 上有 2 个 AI Core。每个 AI Core 中分别包含一个矩阵计算单元，与 NVIDIA GPU 中的 Tensor Core 类似，一个时钟周期可以计算一小块矩阵计算，其支持的数值精度为 INT8 和 FP16（输出的数值精度分别为 INT32 和 FP32）；AI Core 中还包含了一个向量计算单元以及若干标量的计算单元，他们都支持各种精度的整数，以及 FP16 和 FP32。矩阵单元、向量单元和标量单元之间可以通过片上可编程的高速缓存进行数据传输，从而形成对数据处理的流水线。在 AI Core 内有一个 256KB 的统一共享缓存（Unified Buffer）。所有 AI Core 共享 L2 缓存，内存为 8GB 的 LPDDR4X 内存，其带宽为 40GB/s。Ascend310 通过 PCIe3.0×4 的接口插到主机的主板上。

除了最核心的 AI Core 负责 DNN 大量的数值运算，Ascend310 还包括了多个 AI CPU。AI CPU 基于 ARM 的 CPU 小核心架构，其主要有两个用途：一些 AI CPU 的核负责 DNN 中细粒度的张量操作，对于 AI Core 不擅长的运算，可以由 AI CPU 来计算；另一类用途是 AI CPU 上运行着一个微型的操作系统，负责调度 DNN 的计算任务。因此，将 DNN 的任务通过驱动发送到 Ascend310 上之后，就不再需要像 TPU 一样主机的 CPU 负责控制 TPU 执行，而是完全交给 Ascend310 的片上操作系统来进行任务调度，因此可以降低主机 CPU 的工作负载，避免其他应用程序对 DNN 推理的干扰。

要充分利用 NPU 的优势，需要了解 NPU 的具体指令集，并尽可能利用其向量单元和矩阵单元进行计算。NPU 标量指令与 CPU 中常见的指令基本相同，包括了基础运算指令、逻辑指令跳转指令等。向量指令和 CPU 上的单指令多数据（Single Instruction Multi Data，SIMD）指令类似，不同的是其长度比 CPU 上的 SIMD 单元要长很多，且支持 FP16、FP32 和 INT32 类型的运算。向量指令包括向量运算（例如 Add、Max）、向量比较与选择指令（例如 Cmp、Sel）、向量逻辑指令（例如 And）、向量数据搬运（Mov）和一些特殊指令。矩阵乘的指令只支持 INT8 和 FP16 的输入，产生 INT32、FP16 或 FP32 的结果。因此，如果想要应用矩阵乘指令加速数据处理中的通用计算，势必要对数据进行缩放，以避免精度损失甚至溢出。

目前，昇腾处理器还没有办法直接在深度学习框架中直接进行推理，即不能支持算子的即时执行（Eager Execution）。如果需要运行推理首先需要将算子或者计算图通过昇腾张量编译器（Ascend Tensor Compiler，ATC）编译为昇腾的二进制文件，继而在通过昇腾计算语言（Ascend Computing Language, ACL）的运行时加载模型并运行。ACL 同时提供了 C++ 和 Python 的接口，供用户进行内存分配、张量传输、加载执行模型、获取输入输出、性能收集等任务。同时，昇腾工具箱还提供了可以自定义算子的接口，允许用户通过调用细粒度的指令实现具体的算子。本章主要通过 ATC 编译模型并用 ACL 加载执行模型的方式进行计算图的推理。

5.3　性能刻画

表 5-1 NPU 的指令性能测试

Table 5-1 Performance evaluation for basic instructions of NPU

指令类别	输入规模（FP16）	延迟（毫秒）	描述
transpose	1024 × 2048	0.59	对张量进行转置
vec_abs	1024 × 2048	0.21	计算一个张量的绝对值
vec_add	1024 × 2048	0.30	计算两个张量相对应元素相加
vec_max	1024 × 2048	0.30	计算两个张量相对应元素的最大值
vec_and	1024 × 2048	0.41	计算两个张量对应位置元素的逻辑与
vec_cmp	1024 × 2048	0.26	计算两个张量对应位置元素的大小比较
vec_reduce_add	1024 × 2048	0.21	计算一个张量在某一（某些）维度上的累加和
GEMV	m=1,n=1024, k=1024	1.218	计算一个向量与一个矩阵的乘积
matmul	m=1024,n=1024, k=1024	0.80	计算矩阵乘

本节首先通过一组小的测试样例来衡量 Ascend310 的性能，衡量内容包括存储系统的内存带宽、片上 Unified Buffer 带宽；计算单元的计算能力：标量、向量以及矩阵单元的计算性能。这一组测试的目的主要是为了与 CPU 的性能做比较，来衡量 NPU 在数据处理方面的能力。实验平台是华为云的 AI 加速型 ai1.large.4 弹性云服务器。CPU 为 Intel(R) Xeon(R) Gold 6151 CPU 的两个核心，处理器基本频率 3.00 GHz，8GB DDR4 内存，Ubuntu 16.04 操作系统。本节分别从各种指令类型中选择代表性的指令，输入数据的规模目前设置为 1024×2048，这个规模和全高清图片 1920 × 1080 接近，更能反应真实场景下的性能。本实验使用 Ascend 开发套件中的性能收集和测试工具来测量算子在 AI Core 的执行时间。注意，由于衡量性能时，假设数据已经在 NPU 的内存中，因此测量的执行时间不包括将数据从主机内存传输到 NPU 内存的时间。

从表 5-1中可以得到如下的结论：（1）对于转置指令，达到 14.22GB/s。这是因为转置需要细粒度的对每个元素进行操作，需要使用标量指令，计算转置地址等操作，难以达到峰值带宽。(2) 对于向量指令，Ascend310 能够达到接近内存的峰值带宽。对于 vec_abs、vec_add、vec_and、vec_cmp 和 vec_reduce_add，它们达到的内存带宽分别为：38.01GB/s、38.71GB/s、38.71GB/s、29.27GB/s、23.07GB/s 和 19.05GB/s。这说明如果对全高清的图像进行每个元素的处理，Ascend310 能够充分利用内存带宽，达到非常好的理论性能。通常，单个通道 DDR4 内存的带宽可以在 2666MHz 下达到 21.3 GB/s。vec_cmp 没有达到内存峰值带宽，这是因为在 Ascend310 做完向量比较后，比较结果先保存为了 FP16，之后转成了 INT8 类型。vec_reduce_add 由于要进行规约运算，在向量部件中采用二叉树的方式两两相加，因此一条指令需要多拍才能完成，在片上的计算时间也较长。(3) 对于矩阵向量乘法（GEMV），由于其计算仍让为访存密集型，可以计算其内存带宽为 3.44GB/s。这是因为 Ascend 并没有专门面向向量矩阵乘的优化，而是需要先将向量也填充为一个小的矩阵，然后再调用 Ascend310 上矩阵计算单元。这也带

来了性能上的损失。对于规约的长度较大的运算，应该尽可能优化其性能表现。而对于规则的矩阵乘法，可以计算达到了 2.50 TOPS（FP16 精度下）。其计算的性能远超 CPU 的理论峰值计算能力（按照 3.0GHz 计算，AVX-256 指令集理论 FP16 峰值计算性能为 0.48TOPs）。因此，应该利用 NPU 尽可能的计算矩阵运算。

5.4 编程接口与框架设计

在本章中首先展示 GDPNPU 提供的编程接口，之后介绍 GDPNPU 的整体框架设计。由于目前深度学习框架主要采用 Python 语言，因此 GDPNPU 提供了基于 Python 的接口。除了用户可以自定义的接口外，GDPNPU 还提供了若干库函数供编程人员进行调用。在整体框架上，GDPNPU 还是利用了 ATC 和 ACL 的异构编程模型，但是提供了高层级的接口封装，使得程序员从 ACL 繁琐的数据管理与传输、模型管理与加载等复杂的编程接口中解放出来。GDPNPU 还提供了一个运行时系统，以自动进行（1）数据的传输、（2）自动精度变换（3）计算图优化。

5.4.1 编程接口

列表 5-1展示了 GDPNPU 的代码示例。代码中以 TensorFlow 为例构建模型，之后调用 GDPNPU 包提供的 *compile* 接口，即可自动将 TensorFlow 的模型转换为 Ascend310 所支持的模型。在 *compile* 的过程中，GDPNPU 会进行一系列的面向 Ascend310 的精度适配以及性能优化的过程。具体的内容将会在章节 5.5中介绍。之后，调用 *run_asycn* 接口，GDPNPU 会异步地把计算图提交到任务队列中，如果数据在主机，则自动地传输到 Ascend310 上。在最后，*sync* 接口会等待之前提交的计算图完成后返回。GDPNPU 同样支持 PyTorch 模型所构建的网络。

5.4.2 框架设计

如图 5-2所示为 GDPNPU 的整体框架概览。最上层为基于 Python 的编程接口。用户基于 TensorFlow 的 API 编写完成后，调用 *compile* 后，交由 GDPNPU 对计算图进行特定的优化和编译。由于通用数据处理与 DNN 模型的推理在精度方面具有不同的要求，精度自适应模块自动选择满足程序精度要求的算子实现，并插入相应的精度转换算子。如章节 5.3所展示的 Ascend 与 CPU 不同的性能特征，需要针对 Ascend310 的性能特点进行优化，因此进行算子替换，以最大化推理的性能。同时，对数据处理程序通常具有较小的并行度，因此，而 Ascend310 具有很高的并行度，因此需要在多个维度进行并行计算优化。具体的优化选项在章节 5.5进行详细的介绍。

接下来简要介绍 GDPNPU 的运行时系统。运行时主要分为四个部分：

- 模型管理（Model management）。编译优化后的模型由运行时系统管理进行完成存储、加载和下发。经过编译优化后的模型只能在特定的 Ascend 设备上运行，因此，运行时系统会根据底层的设备选择对应版本的模型。由于目前 ACL

```python
import tensorflow as tf
import numpy as np
# The General Data Processing package
import gdp

if __name__=="__main__":
  # Create placeholder for TensorFlow model
  a = tf.keras.Input(shape=(2, 3), batch_size=1, name='a', dtype=tf.float32)
  b = tf.keras.Input(shape=(2, 3), batch_size=1, name='b', dtype=tf.float32)
  # Create the model's compute graph
  c = tf.add(a, b, name='c')

  # Create input and output data
  x = np.ones((2, 3), dtype=np.float32)
  y = np.ones((2, 3), dtype=np.float32)
  z = np.zeros((2, 3), dtype=np.float32)

  # Compile model to ascend
  model = gdp.compile([a, b], [c,])

  # Run the compiled model on NPU
  gdp.run_async(model, {'a':x, 'b': y}, {'c': z}, profile=True)
  # Synchronize for the compute graph to finish
  gdp.sync()

  print(z)
```

图 5-1 GDPNPU 代码示例

Figure 5-1 Code example for GDPNPU

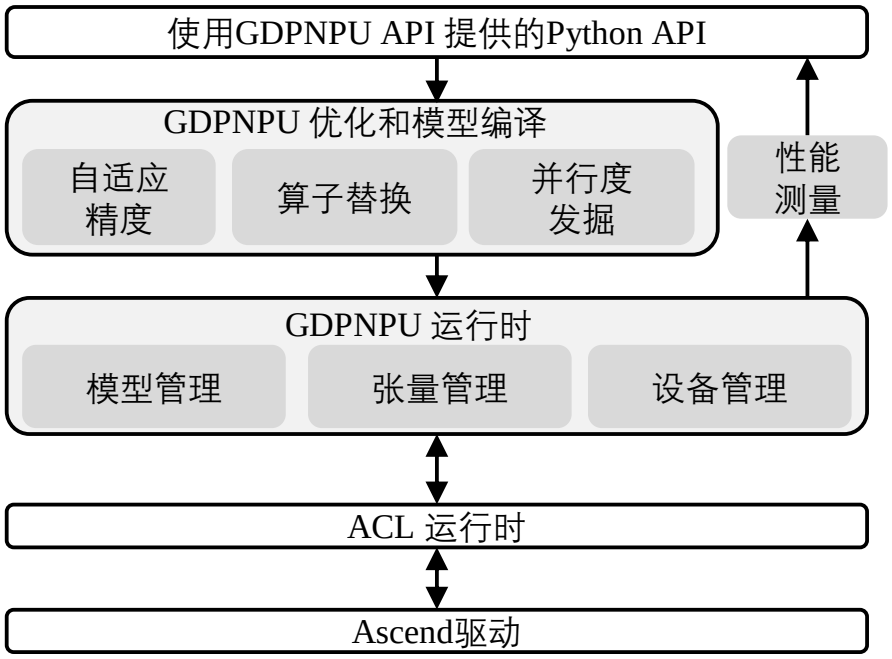


图 5-2 GDPNPU 框架概览

Figure 5-2 System overview of GDPNPU

的限制不支持动态形状，GDPNPU 会为每个不同形状的输入都产生一个对应的优化模型，并在运行的时候根据用户输入选择对应的优化模型。并且如果相同的模型多次被调用，则运行时只会下发一次模型，避免重复加载。

- 张量管理（Tensor management）。由于 TensorFlow 和 PyTorch 都使用 *numpy*[70] 作为与主机进行设备交换的中介，GDPNPU 同样基于 *numpy*。GDPNPU 的运行时会自动根据访问数据的位置，在主机和设备间进行张量传输。
- 设备管理（Device management）。主机上可能会有多张加速卡，用户在运行模型时可以选择具体运行在哪一张卡上。如果没有指定，则默认运行在空闲的设备上。设备的状态通过驱动程序获得。
- 性能收集（Profile）。ACL 提供了细粒度的性能收集接口。主要分为两大类：一类是在主机上 ACL 的 API 执行延迟等、另一类是设备上算子的执行延迟、计算部件利用率等等。GDPNPU 的运行时可以在计算图运行完成后，与 ACL 的运行时和驱动程序交互，自动为用户返回主机和设备上的性能信息，以帮助用户优化计算任务。

5.5 性能优化

5.5.1 图像处理中的控制流处理

数据预处理通常需要相邻元素细粒度的操作。例如，在边缘检测[226] 和光流检测[227] 等任务中都需要判断相邻元素之间的数值大小等操作。NPU 通常为大量的计算设计，难以细粒度地处理相邻元素之间的操作。以一个比较相邻两个元素之间大小，并根据结果设置对应位置的值的图像处理任务，来介绍如何通过张量操作达到相同的计算。本节用 TE 来表达其数学形式。具体关于 TE 的介绍可以参见章节 4.3。如图 5-3所示，代码第 4-5 行描述了如果 *img[i][j]* 的值大于 *img[i-1][j]*，则将该位置设置为 1，否则设置为 0。代码第 8-12 行则是用张量表示的等价替换，其核心在于构造一个 1×3 的卷积核，卷积核与图像进行可分离卷积时，进行如下计算：

```
1  import tvm
2  from tvm import te, tir
3  # Compare the value between two adjacent pixels in horizontal direction
4  mask = te.compute((height, width),
5  lambda i, j: tir.if_then_else(img[i][j]>img[i-1][j], 1, 0))
6
7  # Equally tensor implementation
8  import tensorflow as tf
9  filter = tf.reshape(tf.constant([-1, 1, 0] * 3, tf.float16), [1,3,3,1])
10 out = tf.nn.depthwise_conv2d(img, filter, padding='SAME', strides=1, data_format='NCHW')
11 cmp_out = tf.greater(out, 0)
12 mask = tf.cast(cmp_out, dtype=tf.float16) * tf.constant([1], tf.float16)
```

图 5-3 细粒度元素比较

Figure 5-3 Fine-grained element-wise comparison

$$\begin{bmatrix} img_{i-1,j} & img_{i,j} & img_{i+1,j} \end{bmatrix} \times \begin{bmatrix} -1 & 1 & 0 \end{bmatrix} \Rightarrow img_{i,j} - img_{i-1,j}$$

这样，可以得到 $img_{i,j} - img_{i-1,j}$ 的计算结果，与 0 比较，即可得到真（True）或假（False）。然后将数值转换为 FP16 类型，真值会被转为 1，假值被转为 0，之后与想要的掩膜值相乘，即可得到与使用 *if_then_else* 表达式相同的结果。

算法 3 图像相邻元素掩膜计算的张量代码生成算法

算法 3 Tensor code generation algorithm for image adjacent elements mask computation

Input: An lambda tensor expression: $mask[i][j] = if\_then\_else(\alpha \cdot img[i][j] - \beta \cdot img[i-x][j-y] > K, true\_val, false\_val)$

Result: Code of TensorFlow computation graph

23 $x, y \in \mathbb{Z}^+$

24 $\alpha, \beta, K \in \mathbb{R}$

25 $C, H, W \leftarrow tf.shape(img)$

26 $filter \leftarrow [0]_{2x+1,2y+1,C}$

27 $filter_{x,y,c} \leftarrow \alpha, \quad \forall c < C$

28 $filter_{0,0,c} \leftarrow -\beta, \quad \forall c < C$

29 $code$ = ”mask_true = tf.cast(tf.greater(tf.nn.depthwise_conv2d(img, filter, strides=1, padding=’SAME’), K), tf.float16)
mask_false = tf.cast(tf.less_or_equal(tf.nn.depthwise_conv2d(img, filter, strides=1, padding=’SAME’), K), tf.float16)
true_value_tensor = true_val * mask_true
false_value_tensor = false_val * mask_false
mask = true_value_tensor + false_value_tensor
”

30 **return** *code*

图 30为一个通用的张量代码生成算法，其输入为一个 lambda 表达式，其判断条件为图像的两个元素之间分别与系数 α 和 β 相乘后和 K 比较大小，如果条件为真则返回 $true\_val$，否则返回 $false\_val$。图 30代码第 3 行获取图像的通道数 C，高度 H 以及宽度 W。代码第 4 行初始化一个形状为 $(2x+1, 2y+1, C)$ 的零矩阵，代码第 5-6 行设置卷积核 $filter$ 的的值，由于 GDPNPU 用卷积操作来计算 $\alpha cdot img[i][j] - \beta cdot img[i-x][j-y]$，因此设置卷积核对应位置分别为 α 和 $-\beta$。张量计算的代码 $code$ 的前两行分别计算满足条件判断的掩膜（$mask\_true$）和不满足条件判断的掩膜 $mask\_false$，之后分别与要设置为的值相乘。由于为假的位置值为 0，因此两个掩膜相加即可分别设置对应位置的值。生成的代码会

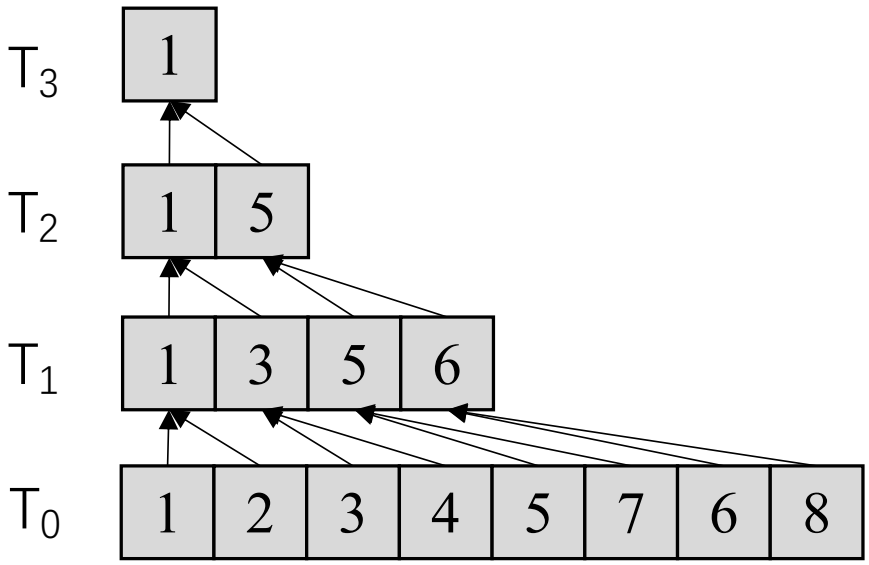


图 5-4 Ascend 310 基于二叉树的 *vec_reduce_min* 指令

Figure 5-4 Binary tree based *vec_reduce_min* instruction on Ascend 310 processor

被输入到 GDPNPU 的运行时中编译为二进制模型。

5.5.2 基于性能测量的算子替换策略

通常，用户表述算法逻辑时并不会考虑算子在硬件上的性能。有时候，可以使用不同的算子（或一系列算子组成的计算图）能够达到相同的计算结果。TASO[228] 对于自动算子替换得到了很好的效果。但是由于在昇腾平台上每次算子替换都需要重新编译以生成模型的二进制文件。对于有 10 个左右的算子，每次编译通常需要 3 至 10 秒钟的时间。基于搜索的自动算子替换会带来很高的时间开销。因此，GDPNPU 在图优化中加入了基于规则的算子替换策略。基于对昇腾芯片基础指令性能基准测试的结果，将一个或一组算子自动替换为另一组算子，从而在保证算法正确性的基础上达到更高的性能。

5.5.2.1 规约算子替换

在数据预处理中，扫描所有数据取得张量中的最大值、最小值或者平均值是非常常见的操作。通常，用户会调用 *reduce_min*、*reduce_max* 和 *reduce_mean* 来完成以上的计算。当规约维度比较小的时候，昇腾的编译器所使用的 *vec_reduce_min* 指令足够高效。如图 5-4所示为 Ascend 310 进行规约计算，返回一个向量中的最小值的过程。其指令基于二叉树实现，假设一条指令规约元素数量是 K，每次规约相邻的两个元素，规约 K 个元素所需要的时间步 $T = log_2(K)$。图 5-4中所示规约 8 个元素，需要 3 个时间步才能完成。规约操作只需要对张量中所有的元素都扫描一遍，但是对于大规模的规约，如章节 5.3中所示，*vec_reduce_sum* 所能达到的带宽只有 19.05GB/s（总带宽为 40GB/s）。因此，对于规约规模较大的，GDPNPU 将规约算子替换为多个算子组成的计算图。如图 5-5所示为将 *re-*

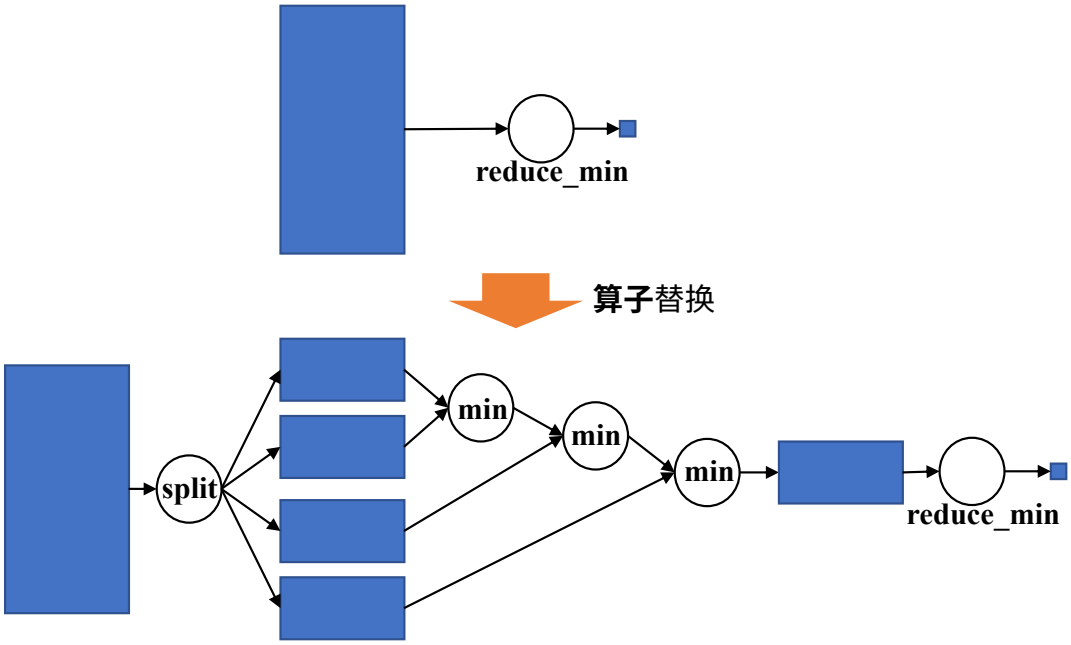


图 5-5 *reduce* 算子的替换

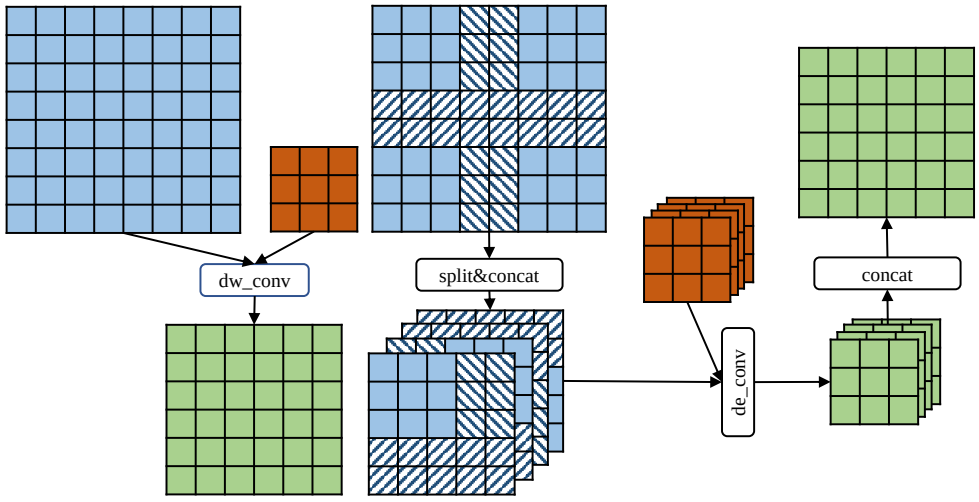
Figure 5-5 Operator substitution for *reduce*

duce_min 替换为由 *split*、*min* 和 *reduce_min* 算子组成成的子图。首先，将张量切分（Split）为 *s* 份，之后 *s* 个切片后的张量基于向量指令实现的 *min* 算子计算按元素的最小值，最后再调用 *reduce_min* 算子做最后的规约。由于 *split* 只是改变了张量的视图，不改变任何数据的排布，因而不带来额外的开销；而 *min* 算子和章节 5.3中 vec_max 一样，可以达到内存最高的带宽，因而能够获得最佳的性能；最后的 *reduce_min* 其规约的规模只相当于原来的 $\frac{1}{s}$，性能损失随之降低。对于一个 1920×1080 分辨率 RGB 的三个通道的图片，寻找每个通道最小值。直接使用 *reduce_min* 的延迟是 1.50 毫秒（达到内存带宽 8.29GB/s），基于图 5-5进行算子替换后的延迟是 0.60 毫秒（达到内存带宽 20.72GB/s），获得 2.5× 倍加速比。

5.5.2.2　图像滤波并行优化

计算机视觉领域有大量的图像滤波运算。例如 OpenCV 中的 *GaussianBlur*、*Soble*、*erode* 和 *Laplacian* 等。图像的滤波操作和 DNN 中的可分离卷积具有相同的计算模式，以下使用可分离卷积指代图像滤波。NPU 加速器中卷积操作主要针对卷积通道数（Channel）数高的优化，具有很高的并行性。而图像处理中期通道数很低，一般为单通道（例如灰度图）、三通道（例如 RGB 彩色图）或者四通道。不能充分发挥智能处理器的算力。

如图所示为针对图像处理领域中图像滤波算子的优化。图 5-6a为对一个单通道的图像做一个 3×3 的卷积。由于只有一个通道，NPU 中不能利用多通道的并行。GDPNPU 对卷积算子做如图 5-6b所示的多通道并行。假设图像的数据布局为 *NCHW*，其中 *N* 代表图像数，*C* 代表图像通道数，*H* 代表图像高度（以像素为单位），*W* 代表图像宽度、首先，将张量切分（*split*）为 *s* 个子块，图中有



(a) 原始的可分离卷积　　(b) 将图像切分增加维度并行

图 5-6 可分离卷积（Depthwise convolution）的算子替换

Figure 5-6 Operator substitution for depthwise convolution

斜线的部分代表需要重复加载的数据元素；之后，将切分后的多个张量在 C 维度进行连接（*concat*）操作，并且将权值也复制 s 份并在 C 维度进行连接。然后进行卷积操作，最后将结果再连接。其中的 s 通过自动调优获得，一般设置为 2 的幂次方，因为昇腾中的矩阵计算单元，其规模就是 2 的幂次方。其中切分操作只需要拷贝连续的内存，因而能够充分利用内存带宽；维度加倍后的张量，能够充分利用 NPU 上的矩阵计算单元，达到更高的计算效率。对于一个 1920×1080 分辨率 RGB 的三个通道的图片，计算高斯模糊（卷积核为大小为 5）。直接使用可分离卷积的延迟是 9.84 毫秒，基于图 5-6算子替换，设置 s 值为 16 时，延迟为 2.29 毫秒，获得 4.30× 倍加速比。

5.5.3 自动数值精度缩放

为了能够减小在低精度下的损失，以达到更高的性能，GDPNPU 可以自动的把数值缩放到合适的精度，以避免在计算过程中出现溢出。Ascend310 的向量单元支持的精度有 FP16、FP32、INT16 和 INT32，矩阵单元支持的精度 INT8 和 FP16。在矩阵单元运算中，如果源操作数为 INT8，输出为 INT32；如果源操作数为 FP16，输出为 FP32。Ascend 的标量单元支持各种精度。由于 INT8 的精度和表示范围都有限，且需要不断的进行数值类型转换，因此主要以 FP16 精度进行运算。根据 IEEE 754 的标准[28]，FP16 由 16 比特组成，包括 1 比特的符号位、5 比特的指数位和 10 比特的尾数位。GDPNPU 主要根据以下的因素来决定缩放的系数 S。（1）输入数据的范围；（2）算子；（3）FP16 的表示精度与范围。对于图像的输入数据，其输入范围固定在 [0, 255] 之间。对于其他输入数据，可以通过调用 *reduce_min* 和 *reduce_max* 获取数据集中的最大值与最小值。本文根据算子的计算性质将其分为了三个类别。GDPNPU 估计算子在最大最小值上运算完达到的最大值，来为算子选择合适的缩放系数。最后，GDPNPU 再根据被缩放

了的算子的系数，将计算图算出来的结果再恢复到高精度的数值。

5.5.3.1　算子类型极其缩放系数的选择

表 5-2 算子的精度敏感分类

Table 5-2 Numerical precision sensitivety classification for operators

敏感程度	包含的算子
不敏感-内存操作	*pad*、*split*、*concat*、*slice*、*expand*、*reshape*、*transpose*、*resize*
不敏感-只返回已有值	*max_pool*、*compare_operator*、*min*、*reduce_min*、*topk*、
敏感-线性运算	*add*、*mul*、*reduce_sum*、*reduce_mean*、*conv*、*matmul*
敏感-非线性运算	*exp*、*sin*、*tanh*、*softmax*、*log*、*sigmoid*、*sqrt* 、*pow* 、*abs*

算子本身的性质会影响计算结果数值的精度。因此，GDPNPU 对算子进行分类，根据算子类别选择是否需要缩放，以及缩放系数如何选择。表 5-2中展示了常用算子的分类。主要分为两大类：不敏感和敏感。不敏感指算子的输出不会造成数值的溢出与精度损失。主要包括两大类：一类是张量的内存操作，主要是重新排布张量的元素，例如 *concat*；另一类是虽然有运算，但是只会返回原有张量中一部分的值，例如 *max_pool*。不敏感算子在分析的时候可以调过，不会造成精度损失或者溢出。敏感的算子也分为了两大类：线性运算和非线性运算。GDPNPU 首先针对线性运算进行数值的缩放。假设算子输入张量中元素的最大值和最小值分别是 $input_{max}$ 和 $input_{min}$，缩放系数一般设置为：

$$S = \frac{1}{rupw2(max(|input_{max}|, |input_{min}|))} \tag{5-1}$$

其中 $rupw2(x)$ 是一个函数，指将值 x 设置到最近的比 x 大的为 2 的整数次幂的数（Round up power of 2）。例如 $rupw2(3) \rightarrow 4$，$rupw2(11) \rightarrow 16$。设置缩放系数到 2 的整数次幂的原因是为了方便进行调试，同时将数值恢复的时候更方便计算。在非极小值情况下（例如指数位全为 0）只会导致 FP16 中的指数位变动，而尾数位保持不变，从而尽可能保证原有的精度。

GDPNPU 为不同的算子设置的规则如下：对于张量加减法（*add*、*sub*）：

$$S = \frac{1}{2 \cdot rupw2(max(|input_{max}|, |input_{min}|))} \tag{5-2}$$

对于规约累加（*reduce_sum*）和规约求平均（*reduce_mean*），在编译模型的时候规约的元素个数为 K，且在编译时已知：

$$S = \frac{1}{K \cdot rupw2(max(|input_{max}|, |input_{min}|))} \tag{5-3}$$

对于按位乘法（mul）：

$$S = \frac{1}{rupw2(max(|input_{max}|, |input_{min}|)^2)} \tag{5-4}$$

对于卷积（$conv$），卷积核的元素个数为 K，：

$$S = \frac{1}{K \cdot rupw2(max(|input_{max}|, |input_{min}|)^2)} \tag{5-5}$$

对于矩阵乘（$conv$），输入矩阵分别为 $M \times K$ 和 $K \times N$：

$$S = \frac{1}{K \cdot rupw2(max(|input_{max}|, |input_{min}|)^2)} \tag{5-6}$$

而对于非线性运算的算子，根据缩放值所得到的结果来推导非缩放值是非平凡的。即假设 f 为非线性运算函数，S 为缩放系数，x 为张量元素值，根据 $f(S \cdot x)$ 计算 $f(x)$ 是很困难的。例如根据 $sin(\frac{x}{16})$ 求 $sin(x)$ 将会引入复杂的计算。因此，如果有非线性运算，则非线性算子之前的算子都不进行数值的缩放。对于开平方根运算（$sqrt$），GDPNPU 做特殊的处理。因为 $\sqrt{S \cdot x} \Leftrightarrow (\sqrt{S} \cdot \sqrt{x})$，可以得到 $\sqrt{x} \leftarrow \frac{\sqrt{S \cdot x}}{\sqrt{S}}$ 因而根据 $\sqrt{S \cdot x}$ 的结果可以很容易的求得 $\sqrt{x}$。

5.5.3.2 自动缩放及数值自动恢复

对于计算图的输入，GDPNPU 首先扫描其中的最大值和最小值，之后根据其第一个算子的计算对数据进行缩放。可以注意到，缩放完的张量，其所有的元素的绝对值都小于 1。之后 GDPNPU 根据输入的范围模拟在算子最大值情况下的计算，按照数据流图计算每个算子会产生的最大值和最小值。如果某个算子会产生溢出，则根据章节 5.5.3.1中的缩放公式进行缩放。在缩放的过程中，算子内记录下当前的缩放因子。

在整个计算图完成推理后，得到 FP16 的结果，将其转换 FP32 的数值。之后反向扫描计算图，将算子中记录的缩放因子相乘并求倒数，即可得到恢复数值，这个过程都是标量运算。将恢复数值与张量再进行一次乘法运算，即可得到最终的结果。最后恢复的过程需要利用张量单元 FP32 精度进行计算。

5.5.3.3 数据扫描以及精度恢复的开销

数据扫描需要发现数据中的最大值和最小值。GDPNPU 在 Ascend310 上扫描一个 $3 \times 1080 \times 1920$ 的图片，只需要 0.6 毫秒，恢复精度需要的 mul 运算也只需要 0.3 毫秒。而 OpenCV 中计算其平均值和方差（cv::meanStdDev）需要 3.47 毫秒，即使是对于访存密集型的运算，这个开销也是可以接受的。而如果是矩阵乘类运算，获得的收益将会更大。

5.6 实验和性能评估

5.6.1　实验平台与工作负载

实验的平台与章节 5.3中描述的一致。在测量性能时，将任务绑定到固定的 CPU 核心上，且后台没有其他计算任务。对比的基准程序为 OpenCV[69] 和 NumPy[70]。OpenCV 是最负盛名的计算机视觉库，包含了非常多的计算机视觉算法，且代码都经过非常精细的优化，本实验中使用的版本是 4.6.0。NumPy 是在深度学习编程中最常使用的数据预处理库，本实验中使用的版本是 1.22.4。工作负载基于 C++ 实现，使用 C++ 库中的 *chrono::steady_clock* 来测量程序执行的延迟。程序会运行 100 次求平均时间。

本实验的应用包括以下六个工作负载：

色调映射（Tone mapping，标记为 *Tone*）[229]。色调映射是图像处理中的重要一步，需要对每个像素应用一个仿射函数。本实验采用了 Academy Color Encoding System[230] 中的色调映射方法。其计算公式为 $f(x) = (x*(a*x+b))/(x*(c*x+d)+e)$。其中 a, b, c, d, e 均为系数，x 为图像的 RGB 的值。

高斯模糊（Gaussian blur，标记为 *Gaussian*）。高斯模糊是进行图像降噪的重要方法。其可以直接被映射为一个可分离卷积操作（Depth-wise convolution）。

计算图片的锐度（Sharpness，标记为 *Sharp*）。照片的锐度是衡量图像质量的中重要指标。例如在 nsvf[231] 到 Nerf 的数据格式转换[232] 中就需要计算图片的锐度。其操作为首先将一张 RGB 彩色图像转为灰度图，之后进行拉普拉斯算子计算图像梯度，再梯度的方差。

图像边缘检测[226]（Canny Edge Detection，标记为 *Canny*）。边缘检测是计算机视觉领域中最经典的任务之一。可以快速的检测图像中的物体的边缘像素点。它由多个阶段组成，包括图像格式转换、高斯滤波、图像梯度计算和极大值抑制等。其中包括卷积运算、细粒度的像素值比较、阈值过滤等操作。其通常在 CPU 或者 GPU 上实现。本实验参考 OpenCV 的实现完成具体的算法。

求数据集均值方差、最大值最小值（Mean 和 Standard deviation，标记为 *Mean-std*）。数据处理中经常需要对数据进行归一化处理，当数据元素数量较多时，会有数值溢出的风险，需要自动对数值进行缩放。需要统计数据集合中的最大最小值。

求 K-近邻（K-Nearest-Neighbor，标记为 *KNN*）。在人脸识别[233] 等图像识别任务经常需要根据当前 DNN 推理的特征向量，在数据库中寻找距离最近的特征向量，以进行特征的匹配。在本实验选择 ImageNet 数据集[162] 在 ResNext101[197] 上推理得到的特征向量，从中随机选择一个特征向量，找出与其最近的一个特征向量。

其中前五个工作负载都是基于 OpenCV 的接口实现，其输入规模为 1080×1920 的全高清图像，最后一个 *KNN* 输入为 5000×1000 的数据集，其中的大小为 5000，特征的维度是 1000。

5.6.2 CPU 单核与 GDPNPU 性能对比

表 5-3 批大小为一的性能对比

Table 5-3 Performance comparation with batch size one

工作负载	NPU-FP32	NPU-FP16	NPU-opt	CPU	SU / NPU-FP32	SU / NPU-FP16	SU / CPU
Tone	5.26	2.94	2.94	2.96	1.79×	1.0×	1.01×
Gaussian	9.25	9.25	4.74	3.93	1.95×	1.95×	0.82×
Sharp	42.82	37.49	25.63	9.29	1.67×	1.44×	0.37×
Canny	440.9	285.74	159.1	88.4	2.77×	1.79×	0.55×
Mean-std	8.74	5.10	3.23	3.49	2.71×	1.58×	1.08×
KNN	4.41	4.41	2.03	13.10	2.2×	2.2×	6.45×

表 5-3为六个工作负载的延迟对比。其中 **NPU-FP32** 表示使用 GDPNPU 的 API 实现，使用 FP32 精度。注意，Ascend 310 的矩阵单元不支持 FP32，当算子被映射到矩阵单元时，切换回默认的 FP16。**NPU-FP16** 表示基于自动精度实现，如章节 5.5.3所述将张量调整为 FP16 并进行数值缩放；**NPU-opt** 表示在 **NPU-FP16** 的基础上进行章节 5.5算子替换优化的延迟。**CPU** 代表使用 OpenCV 或者 NumPy 运行在 CPU 上的延迟。表右半部分为加速比。其中 **SU/NPU-FP32** 代表 **NPU-opt** 相比 **NPU-FP32** 的加速比；**SU/NPU-FP16** 代表 **NPU-opt** 相比 **NPU-opt** 的加速比；**SU/CPU** 代表 **NPU-opt** 相比 **CPU** 的加速比；在六个工作负载上，与 CPU 相比，GDPNPU 最优实现相比 CPU 能够达到 0.37× 到 6.45×。在 *Gaussian*、*Sharp* 和 *Canny* 上的性能比 CPU 都弱，原因主要是因为三个负载都包含卷积操作。在 Ascend310 上，卷积的操作首先需要转置为特定的格式 (具体可参见[59]3.2.1 节)，然后使用矩阵乘实现卷积。特定的格式需要在每次进行卷积操作的前后都进行格式转换，因而带来的额外的较大的开销，导致卷积的性能较差。

NPU-opt 相比 **NPU-FP32** 可以获得 1.0× 到 2.2× 的加速比（平均 1.67×)。通过将 FP32 精度的数据替换为 FP16，每次访问内存加载的数据量减半，且计算单元每次计算的元素数量也翻倍。有两个特殊情况 FP16 相比 FP32 不能获得加速：(1）使用到矩阵计算单元的算子。由于矩阵计算单元只支持 FP16，权值会被提前转为 FP16，因此不会有性能差异，例如 *Gassian*；(2）使用向量单元进行规约累加的算子。在规约累加过程中，为防止溢出会直接用 FP32 类型，也不会有性能差异，例如 *KNN*。**NPU-opt** 相比 **NPU-FP16** 可以获得 1.0× 到 2.77× 的加速比（平均 2.18×)。这是因为通过基于性能测量的算子替换，GDPNPU 可以使用更加高效的算子完成相同的计算任务。

5.6.3 自动精度误差分析

表 5-4展示了在六个工作负载上使用 FP32 精度和 FP16 精度，得到的结果的相对误差。测试集为 ImageNet 数据集。其中 *Canny* 和 *KNN* 的相对误差为 0，这是因为其最终结果都被转换为整数，而他们对应的整数值的结果是相同的。*Tone*、

表 5-4 FP16 与 FP32 结果之间的相对误差

Table 5-4 Ralative error between FP16 and FP32 results

工作负载	*Tone*	*Gaussian*	*Sharp*	*Canny*	*Mean-std*	*KNN*
错误率	0.02%	0.06%	0.18%	0.00%	0.08%	0.00%

Gaussian、*Sharp* 和 *Mean-std* 的结果都是浮点数想比较，其中最大的是 *Sharp*，其误差为 0.18%，这是因为其最后结果为图像所有像素值计算出来的的方差之和。*Sharp* 根据阈值来判断图像是否足够锐利，百分之零点二不到的误差不足以对结果产生影响。其他的误差都在千分之一以内，考虑到最终算出的浮点值会被转换为 0 到 255 的整数，这个误差也在可以接受的范围之内。因此，GDPNPU 的自动转换低精度在提升了性能的同时，仍然保持了结果的正确性与准确性。

5.7　本章小结

本章主要解决在智能处理器上对 DNN 进行数据预处理的问题，提出了一个编程框架 GDPNPU，其为用户提供了在智能处理器上进行数据预处理的编程接口，包含了三项关键的技术：(1) 自动将图像处理中相邻元素的操作变换为智能处理器上的张量算子操作；(2) 自动进行算子替换以优化算子在智能处理器上的性能；(3) 自动的进行低精度的数值替换以降低访存开销，提升处理性能。通过一系列的优化机制，基于 GDPNPU 的编程框架能够有效的在 Ascend310 智能处理器上进行多种数据预处理的任务，甚至传统在 CPU 上才能执行的应用也可以运行在智能处理器上。在六个典型的数据处理任务上，GDPNPU 可将数据处理性能提升最多 6.45×。

第 6 章　总结与展望

6.1　本文工作总结

随着上层 DNN 模型结构越来越复杂、计算量越来越大，而底层的智能处理器其架构也形态各异，为中间的编程框架和编译器的优化带来了极大的挑战。目前的编程框架不能够有效的自动选择最佳的部署策略，不能自动将 DNN 模型的计算映射到异构的计算设备和计算单元上，从而给用户的编程和优化带来了挑战。在编译器的层面，目前的研究工作不能够跨算子的做自动融合和联合优化，限制了编译器的性能优化效果。本文探索了面向深度学习推理应用在编程框架和编译器上的性能优化技术，主要包括以下三方面的创新成果：

为了解决性能和功耗都受限的移动平台上的神经网络性能优化问题，本文提出了一个基于基准测试的自动调优框架 DNNTune 。本文首先通过一系列的基准测试发现，移动平台首先与性能很难有效的执行 DNN 模型特别是参数量较大追求最高准确率的 DNN 模型。为此，DNNTune 设计了两种 DNN 的自动映射策略来优化性能和功耗。第一种是基于端云协同的策略，DNNTune 自动的寻找最佳切分点，并将一部分算子迁移到云端设备协同推理；第二种是基于多计算单元异构并行的策略，DNNTune 基于整数线性规划算法，自动将算子映射到不同计算单元异构并行的执行推理。在不同性能等级的智能手机上的实验表明，与目前最优的自动调优框架相比，DNNTune 最多减少 41.8% 的延迟，节省 15% 的能耗。

为了解决 DNN 算子在 GPU 上的高性能代码生成的问题，本文提出了一个跨算子边界优化 DNN 推理的编译器 Souffle 。当前的 DNN 编译器不能够有效的跨算子边界的发现优化机会，因而不能充分发掘潜在的性能优化机会。Souffle 基于张量表达式的计算图，并在计算图上全局的分析张量的依赖图和复用信息。之后，Souffle 基于数据流分析和启发式算法将整个计算图划分为子图，并在子图上进行数学表达式等价的局部变换，从而找到一个优化的计算核调度，从而提升了指令集的并行和数据复用。本文在英伟达的 A100 GPU 上衡量了 Souffle 执行六个典型 DNN 模型的性能，实验结果表明，Souffle 显著的超过四个当前最优的 DNN 编译器，相比厂商专门优化的 DNN 编译器 TensorRT，Souffle 最高能够达到 5.17× 的加速比。

为了解决 DNN 在智能处理器上的数据预处理编程难和优化难的问题，本文提出了一个面向智能处理器的数据预处理框架 GDPNPU。GDPNPU 提供了一系列简单的编程接口和库，帮助用户快速的在智能处理器上开发针对 DNN 推理数据处理程序，细粒度的处理数据中的元素。更进一步的，GDPNPU 提供了一系列的优化，可以自动的使用低精度数值完成数据预处理，并进行自动算子替换优化数据预处理的性能。在六个典型的数据预处理任务上，GDPNPU 最高能够获得 6.45× 的加速比。

6.2 未来研究展望

近年来，研究人员在算法、框架、编译器和库以及智能处理器等多个软件层次都对对 DNN 推理进行了许多优化。而最近几年深度学习的发展又出现了许多新的趋势。其一，深度学习正在快速的应用到更多的领域中，展现了其惊人的学习和表达能力。例如 AlphaFold[12] 可以进行蛋白质结构的预测，相较于传统的方法大幅提高了准确率；NeRF 类的网络[8] 可以将物体三维信息表示为神经网络，并重新进行视场的渲染；GPT[11] 模型可以根据几句话生成一个小说。这些新模型的参数数量、网络结构都有了很大的变化，为编程框架和编译器带来了新的挑战。例如，GPT-3 模型有 1750 亿的参数，需要多个智能处理器才能够运行一个完整的网络。在编译器与编程框架层面，而 MLIR[133] 正在如火如荼的开发，其为编译器的构建提供了很完善的基础设置，方便研究人员快速的开发编译器和编程框架，并与多种前端和后端结合到一起。JAX[234] 具有强大的自动微分能力和优异的性能正在获得广泛的关注与应用。如何优化编程框架与编译器，支撑前沿的深度学习模型新的特性，充分利用底层智能处理器的计算能力，是系统研究人员需要不断探究的问题。

结合本文在编程和编译器对 DNN 推理的优化问题，我们继续围绕着这个研究方向展开研究，主要包括以下几个方面：

(1) DNN 模型在多 GPU 上的协同推理优化。本文提出了 DNN 映射方法针对的是参数量较小的模型，如何将超大规模的模型映射到多个 GPU 上并行的执行推理是一个新的问题。其中需要涉及到 DNN 算子的映射策略、计算核之间高效的同步、多个 GPU 上如何进行算子融合以及编译器如何优化跨 GPU 的 kernel 等问题。我们将在未来的研究工作中针对新型超大规模 DNN 在多 GPU 上的推理优化展开研究。

(2) 面向多算子代码生成的性能代价模型。在第 4章中，Souffle 将 TE 编译为二进制，然后从中抽取出占用的共享内存数量等信息。在后端，依赖于 Ansor 搜索调度继而进行代码生成。而目前的工作都是针对单算子进行性能调优。在未来的工作中，将研究如何对多个 TE 和智能处理器来构建性能代价模型，打破算子边界，同时对多个算子进行调度搜索、指令调度和代码生成，从而实现多个算子整体整体性能的最优。

(3) 面向自动微分框架的自动算子融合。本文的工作主要面向 DNN 推理过程进行算子融合，并且是离线的进行性能调优。在自动微分框架中，会存在反向微分以及权值更新的计算，如果能够对这两个过程也进行算子融合，就有可能大大缩短训练时间。更进一步的，自动微分框架需要进行即时编译（Just-in-Time Compilation），要求自动调优的时间要在分钟甚至秒级别完成。因此，探究如何对反向传播和权值更新的算子进行算子融合优化，并结合性能代价模型优化即时编译的产生的代码性能，将会是未来工作的一个方向。

参考文献

[1] Simonyan K, Zisserman A. Very deep convolutional networks for large-scale image recognition [C]//Bengio Y, LeCun Y. 3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings. 2015.

[2] Szegedy C, Liu W, Jia Y, et al. Going deeper with convolutions [C]//IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015. IEEE Computer Society, 2015: 1-9.

[3] He K, Zhang X, Ren S, et al. Deep residual learning for image recognition [C]//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016: 770-778.

[4] Liu C, Zoph B, Neumann M, et al. Progressive neural architecture search [C]//Ferrari V, Hebert M, Sminchisescu C, et al. Computer Vision – ECCV 2018. Cham: Springer International Publishing, 2018: 19-35.

[5] Tan M, Le Q V. Efficientnet: Rethinking model scaling for convolutional neural networks [C]//Chaudhuri K, Salakhutdinov R. Proceedings of Machine Learning Research: volume 97 Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA. PMLR, 2019: 6105-6114.

[6] Hochreiter S, Schmidhuber J. Long short-term memory [J]. Neural Comput., 1997, 9(8): 1735-1780.

[7] Hannun A Y, Case C, Casper J, et al. Deep speech: Scaling up end-to-end speech recognition [J]. CoRR, 2014, abs/1412.5567.

[8] Mildenhall B, Srinivasan P P, Tancik M, et al. Nerf: Representing scenes as neural radiance fields for view synthesis [C]//Vedaldi A, Bischof H, Brox T, et al. Lecture Notes in Computer Science: volume 12346 Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part I. Springer, 2020: 405-421.

[9] Vaswani A, Shazeer N, Parmar N, et al. Attention is all you need [C]//Guyon I, von Luxburg U, Bengio S, et al. Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA. 2017: 5998-6008.

[10] Devlin J, Chang M, Lee K, et al. BERT: pre-training of deep bidirectional transformers for language understanding [C]//Burstein J, Doran C, Solorio T. Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers). Association for Computational Linguistics, 2019: 4171-4186.

[11] Brown T B, Mann B, Ryder N, et al. Language models are few-shot learners [C]//Larochelle H, Ranzato M, Hadsell R, et al. Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual: volume 33. 2020: 1877-1901.

[12] Jumper J, Evans R, Pritzel A, et al. Highly accurate protein structure prediction with alphafold [J]. Nature, 2021, 596(7873): 583-589.

[13] Wang J, Huang P, Zhao H, et al. Billion-scale commodity embedding for e-commerce recommendation in alibaba [C]//KDD '18: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. New York, NY, USA: Association for Computing Machinery, 2018: 839–848.

[14] Zhou G, Mou N, Fan Y, et al. Deep interest evolution network for click-through rate prediction [C]//The Thirty-Third AAAI Conference on Artificial Intelligence, AAAI 2019, The Thirty-First Innovative Applications of Artificial Intelligence Conference, IAAI 2019, The Ninth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2019, Honolulu, Hawaii, USA, January 27 - February 1, 2019. AAAI Press, 2019: 5941-5948.

[15] Xu M, Liu J, Liu Y, et al. A first look at deep learning apps on smartphones [C]//WWW '19: The World Wide Web Conference. New York, NY, USA: Association for Computing Machinery, 2019: 2125–2136.

[16] Lv C, Niu C, Gu R, et al. Walle: An End-to-End, General-Purpose, and Large-Scale production system for Device-Cloud collaborative machine learning [C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 249-265.

[17] LeCun Y, Boser B, Denker J S, et al. Backpropagation applied to handwritten zip code recognition [J]. Neural Computation, 1989, 1: 541-551.

[18] Ash Turner. October 2022 mobile user statistics: Discover the number of phones in the world & smartphone penetration by country or region. [EB/OL]. 2022. https://www.bankmycell.com/blog/how-many-phones-are-in-the-world.

[19] Wu C, Brooks D, Chen K, et al. Machine learning at facebook: Understanding inference at the edge [C]//25th IEEE International Symposium on High Performance Computer Architecture, HPCA 2019, Washington, DC, USA, February 16-20, 2019. IEEE, 2019: 331-344.

[20] Hazelwood K, Bird S, Brooks D, et al. Applied machine learning at facebook: A datacenter infrastructure perspective [C]//2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). 2018: 620-629.

[21] Hu Q, Sun P, Yan S, et al. Characterization and prediction of deep learning workloads in large-scale gpu datacenters [C]//SC '21: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. New York, NY, USA: Association for Computing Machinery, 2021.

[22] Einav Mor-Samuels. Improving app performance (and why it' s so important) [EB/OL]. 2022. https://www.appsflyer.com/blog/tips-strategy/app-performance/.

[23] Hestness J, Ardalani N, Diamos G F. Beyond human-level accuracy: computational challenges in deep learning [C]//Hollingsworth J K, Keidar I. Proceedings of the 24th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP 2019, Washington, DC, USA, February 16-20, 2019. ACM, 2019: 1-14.

[24] Chen Y, Luo T, Liu S, et al. Dadiannao: A machine-learning supercomputer [C]//MICRO-47: Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture. USA: IEEE Computer Society, 2014: 609–622.

[25] Jouppi N P, Young C, Patil N, et al. In-datacenter performance analysis of a tensor processing unit [C]//ISCA '17: Proceedings of the 44th Annual International Symposium on Computer Architecture. New York, NY, USA: Association for Computing Machinery, 2017: 1-12.

[26] NVIDIA Corporation. Nvidia tesla v100 gpu architecture [EB/OL]. 2017. https://www.nvidia.com/en-us/data-center/volta-gpu-architecture/.

[27] NVIDIA Corporation. Nvidia ampere architecture in-depth [EB/OL]. 2020. https://developer.nvidia.com/blog/nvidia-ampere-architecture-in-depth/.

[28] Ieee standard for floating-point arithmetic [J]. IEEE Std 754-2008, 2008: 1-70.

[29] Krizhevsky A, Sutskever I, Hinton G E. Imagenet classification with deep convolutional neural networks [M]//NIPS'12: Proceedings of the 25th International Conference on Neural Information Processing Systems - Volume 1. Lake Tahoe, Nevada, 2012: 1097-1105.

[30] Zhu M, Gupta S. To prune, or not to prune: Exploring the efficacy of pruning for model compression [C]//6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Workshop Track Proceedings. OpenReview.net, 2018.

[31] Niu W, Ma X, Lin S, et al. Patdnn: Achieving real-time DNN execution on mobile devices with pattern-based weight pruning [C]//Larus J R, Ceze L, Strauss K. ASPLOS '20: Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, March 16-20, 2020 [ASPLOS 2020 was canceled because of COVID-19]. ACM, 2020: 907-922.

[32] Gupta S, Agrawal A, Gopalakrishnan K, et al. Deep learning with limited numerical precision [C]//Bach F R, Blei D M. JMLR Workshop and Conference Proceedings: volume 37 Proceedings of the 32nd International Conference on Machine Learning, ICML 2015, Lille, France, 6-11 July 2015. JMLR.org, 2015: 1737-1746.

[33] Howard A G, Zhu M, Chen B, et al. Mobilenets: Efficient convolutional neural networks for mobile vision applications [J]. CoRR, 2017, abs/1704.04861.

[34] Sandler M, Howard A G, Zhu M, et al. Mobilenetv2: Inverted residuals and linear bottlenecks [C]//2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018. Computer Vision Foundation / IEEE Computer Society, 2018: 4510-4520.

[35] Iandola F N, Moskewicz M W, Ashraf K, et al. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size [J]. CoRR, 2016, abs/1602.07360.

[36] Hinton G E, Vinyals O, Dean J. Distilling the knowledge in a neural network [J]. CoRR, 2015, abs/1503.02531.

[37] Zoph B, Le Q V. Neural architecture search with reinforcement learning [C]//5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net, 2017.

[38] Abadi M, Barham P, Chen J, et al. Tensorflow: A system for large-scale machine learning [C]//12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16). Savannah, GA: USENIX Association, 2016: 265-283.

[39] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library [C]//Wallach H M, Larochelle H, Beygelzimer A, et al. Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, 8-14 December 2019, Vancouver, BC, Canada. 2019: 8024-8035.

[40] Google Inc. Tensorflow lite is for mobile and embedded devices. [EB/OL]. 2018. https://www.tensorflow.org/lite/.

[41] Mobile ai compute engine documentation [EB/OL]. 2018. https://mace.readthedocs.io/en/latest/index.html.

[42] Jiang X, Wang H, Chen Y, et al. MNN: A universal and efficient inference engine [C]// Dhillon I S, Papailiopoulos D S, Sze V. Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020. mlsys.org, 2020.

[43] Tensorflow lite [EB/OL]. 2018[2018-04-07]. https://www.tensorflow.org/mobile/tflite/.

[44] Janapa Reddi V, Kanter D, Mattson P, et al. Mlperf mobile inference benchmark: An industry-standard open-source machine learning benchmark for on-device ai [C]//Marculescu D, Chi Y, Wu C. Proceedings of Machine Learning and Systems: volume 4. 2022: 352-369.

[45] Larsen R M, Shpeisman T. Tensorflow graph optimizations [EB/OL]. 2019. https://www.tensorflow.org/guide/graph_optimization.

[46] Jia Z, Padon O, Thomas J, et al. Taso: Optimizing deep learning computation with automatic generation of graph substitutions [C]//SOSP ' 19: Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2019: 47–62.

[47] Kang Y, Hauswald J, Gao C, et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge [C]//Chen Y, Temam O, Carter J. Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. ACM, 2017: 615-629.

[48] Kim Y, Kim J, Chae D, et al. μlayer: Low latency on-device inference using cooperative single-layer acceleration and processor-friendly quantization [C]//Candea G, van Renesse R, Fetzer C. Proceedings of the Fourteenth EuroSys Conference 2019, Dresden, Germany, March 25-28, 2019. ACM, 2019: 45:1-45:15.

[49] Han M, Hyun J, Park S, et al. Mosaic: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference [C]//2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT). 2019: 165-177.

[50] Sabne A. Xla : Compiling machine learning for peak performance [Z]. 2020.

[51] Chen T, Moreau T, Jiang Z, et al. TVM: An automated end-to-end optimizing compiler for deep learning [C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 578-594.

[52] Cyphers S, Bansal A K, Bhiwandiwalla A, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning [J]. CoRR, 2018, abs/1801.08058.

[53] Li M, Liu Y, Liu X, et al. The deep learning compiler: A comprehensive survey [J]. IEEE Trans. Parallel Distributed Syst., 2021, 32(3): 708-727.

[54] Chetlur S, Woolley C, Vandermersch P, et al. cudnn: Efficient primitives for deep learning [M]. arXiv, 2014.

[55] arm comepute library [EB/OL]. 2021[2021-05-20]. https://github.com/ARM-software/ComputeLibrary.

[56] Liu D, Chen T, Liu S, et al. Pudiannao: A polyvalent machine learning accelerator [C]// ASPLOS '15: Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2015: 369–381.

[57] Liu S, Du Z, Tao J, et al. Cambricon: An instruction set architecture for neural networks [J]. SIGARCH Comput. Archit. News, 2016, 44(3): 393-405.

[58] olive.zhao. Ascend ai computing engine -da vinci architecture [Z]. 2021.

[59] 华为智能计算技术丛书: 昇腾 AI 处理器架构与编程: 深入理解 CANN 技术原理及应用 [M]. 清华大学出版社, 2019.

[60] NVIDIA Corporation. Cuda toolkit | nvidia developer [EB/OL]. (2021-07-25). https://developer.nvidia.com/cuda-toolkit.

[61] Wang H, Zhai J, Gao M, et al. Pet: Optimizing tensor programs with partially equivalent transformations and automated corrections [C]//15th USENIX Symposium on Operating Systems Design and Implementation (OSDI 21). 2021: 37-54.

[62] Ma L, Xie Z, Yang Z, et al. Rammer: Enabling holistic deep learning compiler optimizations with rtasks [C]//14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20). USENIX Association, 2020: 881-897.

[63] Chen T, Zheng L, Yan E Q, et al. Learning to optimize tensor programs [C]//Bengio S, Wallach H M, Larochelle H, et al. Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada. 2018: 3393-3404.

[64] Zheng S, Liang Y, Wang S, et al. Flextensor: An automatic schedule exploration and optimization framework for tensor computation on heterogeneous system [C]//ASPLOS '20: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2020: 859-873.

[65] Zhu H, Wu R, Diao Y, et al. ROLLER: Fast and efficient tensor compilation for deep learning [C]//16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22). Carlsbad, CA: USENIX Association, 2022: 233-248.

[66] Lei Zhang, Feng Dong, Yuping Zhao, Fiona Zhao, Ying Hu, Jason Wang. Optimization practice of deep learning inference deployment on intel processors [EB/OL]. (2019-02-19). https://www.intel.com/content/www/us/en/developer/articles/technical/optimization-practice-of-deep-learning-inference-deployment-on-intel-processors.html.

[67] Kang D, Mathur A, Veeramacheneni T, et al. Jointly optimizing preprocessing and inference for dnn-based visual analytics [J]. Proc. VLDB Endow., 2020, 14(2): 87-100.

[68] Isenko A, Mayer R, Jedele J, et al. Where is my training bottleneck? hidden trade-offs in deep learning preprocessing pipelines [C]//SIGMOD '22: Proceedings of the 2022 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2022: 1825–1839.

[69] Bradski G. The OpenCV Library [J]. Dr. Dobb's Journal of Software Tools, 2000.

[70] Harris C R, Millman K J, van der Walt S J, et al. Array programming with NumPy [J]. Nature, 2020, 585(7825): 357-362.

[71] Xu M, Liu J, Liu Y, et al. A first look at deep learning apps on smartphones [C]//Liu L, White R W, Mantrach A, et al. The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019. ACM, 2019: 2125-2136.

[72] Hadidi R, Cao J, Xie Y, et al. Characterizing the deployment of deep neural networks on commercial edge devices [C]//IEEE International Symposium on Workload Characterization, IISWC 2019, Orlando, FL, USA, November 3-5, 2019. IEEE, 2019: 35-48.

[73] Ignatov A, Timofte R, Chou W, et al. AI benchmark: Running deep neural networks on android smartphones [C]//Leal-Taixé L, Roth S. Lecture Notes in Computer Science: volume 11133 Computer Vision - ECCV 2018 Workshops - Munich, Germany, September 8-14, 2018, Proceedings, Part V. Springer, 2018: 288-314.

[74] Turner J, Cano J, Radu V, et al. Characterising across-stack optimisations for deep convolutional neural networks [C]//2018 IEEE International Symposium on Workload Characteri-

zation, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018. IEEE Computer Society, 2018: 101-110.

[75] Merck M L, Wang B, Liu L, et al. Characterizing the execution of deep neural networks on collaborative robots and edge devices [C]//Furlani T R. Proceedings of the Practice and Experience in Advanced Research Computing on Rise of the Machines (learning), PEARC 2019, Chicago, IL, USA, July 28 - August 01, 2019. ACM, 2019: 65:1-65:6.

[76] Ogden S S, Guo T. Characterizing the deep neural networks inference performance of mobile applications [J]. CoRR, 2019, abs/1909.04783.

[77] Bianco S, Cadène R, Celona L, et al. Benchmark analysis of representative deep neural network architectures [J]. IEEE Access, 2018, 6: 64270-64277.

[78] Introducing the eembc mlmark benchmark. [EB/OL]. 2019. https://www.eembc.org/mlmark/index.php.

[79] Reddi V J, Cheng C, Kanter D, et al. Mlperf inference benchmark [C]//47th ACM/IEEE Annual International Symposium on Computer Architecture, ISCA 2020, Valencia, Spain, May 30 - June 3, 2020. IEEE, 2020: 446-459.

[80] Hanhirova J, Kämäräinen T, Seppälä S, et al. Latency and throughput characterization of convolutional neural networks for mobile computer vision [C]//César P, Zink M, Murray N. Proceedings of the 9th ACM Multimedia Systems Conference, MMSys 2018, Amsterdam, The Netherlands, June 12-15, 2018. ACM, 2018: 204-215.

[81] Kim H, Nam H, Jung W, et al. Performance analysis of CNN frameworks for gpus [C]//2017 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2017, Santa Rosa, CA, USA, April 24-25, 2017. IEEE Computer Society, 2017: 55-64.

[82] Karki A, Keshava C P, Shivakumar S M, et al. Tango: A deep neural network benchmark suite for various accelerators [C]//IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS 2019, Madison, WI, USA, March 24-26, 2019. IEEE, 2019: 137-138.

[83] Turner J, Cano J, Radu V, et al. Characterising across-stack optimisations for deep convolutional neural networks [C]//2018 IEEE International Symposium on Workload Characterization (IISWC). 2018: 101-110.

[84] Wang Y, Wei G, Brooks D. Benchmarking tpu, gpu, and CPU platforms for deep learning [J]. CoRR, 2019, abs/1907.10701.

[85] Zhu H, Akrout M, Zheng B, et al. Benchmarking and analyzing deep neural network training [C]//2018 IEEE International Symposium on Workload Characterization, IISWC 2018, Raleigh, NC, USA, September 30 - October 2, 2018. IEEE Computer Society, 2018: 88-100.

[86] Dong S, Kaeli D. Dnnmark: A deep neural network benchmark suite for gpus [C]//GPGPU-10: Proceedings of the General Purpose GPUs. New York, NY, USA: Association for Computing Machinery, 2017: 63–72.

[87] Ren Y, Yoo S, Hoisie A. Performance analysis of deep learning workloads on leading-edge systems [C]//2019 IEEE/ACM Performance Modeling, Benchmarking and Simulation of High Performance Computer Systems (PMBS). 2019: 103-113.

[88] Alibaba Inc. Ai matrix [EB/OL]. 2018. https://aimatrix.ai/en-us/.

[89] Cuervo E, Balasubramanian A, Cho D k, et al. Maui: making smartphones last longer with code offload [C]//MobiSys'10. ACM, 2010: 49-62.

[90] Chun B G, Ihm S, Maniatis P, et al. Clonecloud: elastic execution between mobile device and cloud [C]//EuroSys'11. ACM, 2011: 301-314.

[91] Gordon M S, Jamshidi D A, Mahlke S A, et al. Comet: Code offload by migrating execution transparently. [C]//OSDI: volume 12. 2012: 93-106.

[92] Kang Y, Hauswald J, Gao C, et al. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge [C]//Chen Y, Temam O, Carter J. Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017. ACM, 2017: 615-629.

[93] Mao Y, You C, Zhang J, et al. A survey on mobile edge computing: The communication perspective [J]. IEEE Commun. Surv. Tutorials, 2017, 19(4): 2322-2358.

[94] Teerapittayanon S, McDanel B, Kung H T. Distributed deep neural networks over the cloud, the edge and end devices [C]//Lee K, Liu L. 37th IEEE International Conference on Distributed Computing Systems, ICDCS 2017, Atlanta, GA, USA, June 5-8, 2017. IEEE Computer Society, 2017: 328-339.

[95] Das R B, Bozdog N V, Makkes M X, et al. Kea: A computation offloading system for smartphone sensor data [C]//IEEE International Conference on Cloud Computing Technology and Science, CloudCom 2017, Hong Kong, December 11-14, 2017. IEEE Computer Society, 2017: 9-16.

[96] Jeong H J, Lee H J, Shin C H, et al. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers [C]//SoCC ' 18: Proceedings of the ACM Symposium on Cloud Computing. New York, NY, USA: Association for Computing Machinery, 2018: 401–411.

[97] Shin K Y, Jeong H J, Moon S M. Enhanced partitioning of dnn layers for uploading from mobile devices to edge servers [C]//EMDL ' 19: The 3rd International Workshop on Deep Learning for Mobile Systems and Applications. New York, NY, USA: Association for Computing Machinery, 2019: 35–40.

[98] Fang Z, Hong D, Gupta R K. Serving deep neural networks at the cloud edge for vision applications on mobile platforms [C]//Zink M, Toni L, Begen A C. Proceedings of the 10th ACM Multimedia Systems Conference, MMSys 2019, Amherst, MA, USA, June 18-21, 2019. ACM, 2019: 36-47.

[99] Zeng L, Li E, Zhou Z, et al. Boomerang: On-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things [J]. IEEE Network, 2019, 33 (5): 96-103.

[100] Fang Z, Lin J, Srivastava M B, et al. Multi-tenant mobile offloading systems for real-time computer vision applications [C]//Hansdah R C, Krishnaswamy D, Vaidya N. Proceedings of the 20th International Conference on Distributed Computing and Networking, ICDCN 2019, Bangalore, India, January 04-07, 2019. ACM, 2019: 21-30.

[101] Zhang J, Letaief K B. Mobile edge intelligence and computing for the internet of vehicles [J]. Proceedings of the IEEE, 2020, 108(2): 246-261.

[102] Fang Y, Jin Z, Zheng R. Teamnet: A collaborative inference framework on the edge [C]//39th IEEE International Conference on Distributed Computing Systems, ICDCS 2019, Dallas, TX, USA, July 7-10, 2019. IEEE, 2019: 1487-1496.

[103] Funai C, Tapparello C, Heinzelman W. Computational offloading for energy constrained devices in multi-hop cooperative networks [J]. IEEE Transactions on Mobile Computing, 2020, 19(1): 60-73.

[104] Xu M, Qian F, Zhu M, et al. Deepwear: Adaptive local offloading for on-wearable deep learning [J]. IEEE Transactions on Mobile Computing, 2020, 19(2): 314-330.

[105] Chatzopoulos D, Bermejo C, Kosta S, et al. Offloading computations to mobile devices and cloudlets via an upgraded nfc communication protocol [J]. IEEE Transactions on Mobile Computing, 2020, 19(3): 640-653.

[106] Zhao P, Tian H, Chen K, et al. Context-aware tdd configuration and resource allocation for mobile edge computing [J]. IEEE Transactions on Communications, 2020, 68(2): 1118-1131.

[107] Han S, Mao H, Dally W J. Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding [C]//Bengio Y, LeCun Y. 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings. 2016.

[108] Loc H N, Lee Y, Balan R K. Deepmon: Mobile gpu-based deep learning framework for continuous vision applications [C]//Choudhury T, Ko S Y, Campbell A, et al. Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services, MobiSys'17, Niagara Falls, NY, USA, June 19-23, 2017. ACM, 2017: 82-95.

[109] Tencent-ncnn Developers. ncnn is a high-performance neural network inference framework optimized for the mobile platform [EB/OL]. 2020. https://github.com/Tencent/ncnn.

[110] Developers P L. Paddle lite [EB/OL]. 2020. https://paddle-lite.readthedocs.io/zh/latest/.

[111] Chen T, Moreau T, Jiang Z, et al. Tvm: An automated end-to-end optimizing compiler for deep learning [C]//OSDI'18: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation. USA: USENIX Association, 2018: 579–594.

[112] Mirhoseini A, Pham H, Le Q V, et al. Device placement optimization with reinforcement learning [C]//Precup D, Teh Y W. Proceedings of Machine Learning Research: volume 70 Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017. PMLR, 2017: 2430-2439.

[113] Gao Y, Chen L, Li B. Post: Device placement with cross-entropy minimization and proximal policy optimization [C]//Bengio S, Wallach H M, Larochelle H, et al. Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, 3-8 December 2018, Montréal, Canada. 2018: 9993-10002.

[114] Demirci G, Marincic I, Hoffmann H. A divide and conquer algorithm for DAG scheduling under power constraints [C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC 2018, Dallas, TX, USA, November 11-16, 2018. IEEE / ACM, 2018: 36:1-36:12.

[115] Han M, Hyun J, Park S, et al. MOSAIC: heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference [C]//28th International Conference on Parallel Architectures and Compilation Techniques, PACT 2019, Seattle, WA, USA, September 23-26, 2019. IEEE, 2019: 165-177.

[116] Wang S, Yang P, Zheng Y, et al. Horizontally fused training array: An effective hardware utilization squeezer for training novel deep learning models [J]. Proceedings of Machine Learning and Systems, 2021, 3: 599-623.

[117] Roesch J, Lyubomirsky S, Weber L, et al. Relay: a new IR for machine learning frameworks [C]//Gottschlich J, Cheung A. Proceedings of the 2nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages, MAPL@PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018. ACM, 2018: 58-68.

[118] Niu W, Guan J, Wang Y, et al. Dnnfusion: Accelerating deep neural networks execution with advanced operator fusion [C]//PLDI 2021: Proceedings of the 42nd ACM SIGPLAN

International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2021: 883–898.

[119] Jung W, Dao T T, Lee J. Deepcuts: a deep learning optimization framework for versatile gpu workloads [C]//Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. 2021: 190-205.

[120] Zheng Z, Yang X, Zhao P, et al. Astitch: Enabling a new multi-dimensional optimization space for memory-intensive ml training and inference on modern simt architectures [C]// ASPLOS 2022: Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. New York, NY, USA: Association for Computing Machinery, 2022: 359–373.

[121] Zhao J, Gao X, Xia R, et al. Apollo: Automatic partition-based operator fusion through layer by layer optimization [C]//Marculescu D, Chi Y, Wu C. Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org, 2022.

[122] Jangda A, Bondhugula U. An effective fusion and tile size model for optimizing image processing pipelines [C]//PPoPP '18: Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2018: 261–275.

[123] Mehta S, Lin P H, Yew P C. Revisiting loop fusion in the polyhedral framework [C]//PPoPP '14: Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. New York, NY, USA: Association for Computing Machinery, 2014: 233–246.

[124] Bondhugula U, Günlük O, Dash S, et al. A model for fusion and code motion in an automatic parallelizing compiler [C]//Salapura V, Gschwind M, Knoop J. 19th International Conference on Parallel Architectures and Compilation Techniques, PACT 2010, Vienna, Austria, September 11-15, 2010. ACM, 2010: 343-352.

[125] Qasem A, Kennedy K. Profitable loop fusion and tiling using model-driven empirical search [C]//Egan G K, Muraoka Y. Proceedings of the 20th Annual International Conference on Supercomputing, ICS 2006, Cairns, Queensland, Australia, June 28 - July 01, 2006. ACM, 2006: 249-258.

[126] Kjolstad F, Chou S, Lugato D, et al. Taco: A tool to generate tensor algebra kernels [C]// 2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE, 2017: 943-948.

[127] Baghdadi R, Ray J, Romdhane M B, et al. Tiramisu: A polyhedral compiler for expressing fast and portable code [C]//2019 IEEE/ACM International Symposium on Code Generation and Optimization (CGO). 2019: 193-205.

[128] Chen T, Moreau T, Jiang Z, et al. TVM: An automated end-to-end optimizing compiler for deep learning [C]//13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18). Carlsbad, CA: USENIX Association, 2018: 578-594.

[129] Vasilache N, Zinenko O, Theodoridis T, et al. Tensor comprehensions: Framework-agnostic high-performance machine learning abstractions [J]. CoRR, 2018, abs/1802.04730.

[130] Rotem N, Fix J, Abdulrasool S, et al. Glow: Graph lowering compiler techniques for neural networks [J]. CoRR, 2018, abs/1805.00907.

[131] NVIDIA Corporation. Tensorrt [EB/OL]. 2021. https://developer.nvidia.com/tensorrt.

[132] Ikarashi Y, Bernstein G L, Reinking A, et al. Exocompilation for productive programming of hardware accelerators [C]//PLDI 2022: Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2022: 703–718.

[133] Lattner C, Pienaar J A, Amini M, et al. MLIR: A compiler infrastructure for the end of moore's law [J]. CoRR, 2020, abs/2002.11054.

[134] Ragan-Kelley J, Barnes C, Adams A, et al. Halide: A language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines [C]//PLDI '13: Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2013: 519-530.

[135] Chen T, Guestrin C. XGBoost: A scalable tree boosting system [C]//KDD '16: Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining. New York, NY, USA: ACM, 2016: 785-794.

[136] Zheng L, Jia C, Sun M, et al. Ansor: Generating high-performance tensor programs for deep learning [C]//14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020. USENIX Association, 2020: 863-879.

[137] Kuchnik M, Klimovic A, Simsa J, et al. Plumber: Diagnosing and removing performance bottlenecks in machine learning data pipelines [C]//Marculescu D, Chi Y, Wu C. Proceedings of Machine Learning and Systems 2022, MLSys 2022, Santa Clara, CA, USA, August 29 - September 1, 2022. mlsys.org, 2022.

[138] Umesh P. Image processing in python [J]. CSI Communications, 2012, 23.

[139] Riba E, Mishkin D, Ponsa D, et al. Kornia: an open source differentiable computer vision library for pytorch [J]. CoRR, 2019, abs/1910.02190.

[140] Paszke A, Gross S, Massa F, et al. Pytorch: An imperative style, high-performance deep learning library [C]//Wallach H M, Larochelle H, Beygelzimer A, et al. Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada. 2019: 8024-8035.

[141] Google Inc. Module: tf.image [EB/OL]. 2022. https://www.tensorflow.org/api_docs/python/tf/image.

[142] Marcel S, Rodriguez Y. Torchvision the machine-vision package of torch [C]//MM '10: Proceedings of the 18th ACM International Conference on Multimedia. New York, NY, USA: Association for Computing Machinery, 2010: 1485–1488.

[143] Pérez-García F, Sparks R, Ourselin S. Torchio: A python library for efficient loading, preprocessing, augmentation and patch-based sampling of medical images in deep learning [J]. Computer Methods and Programs in Biomedicine, 2021, 208.

[144] Gibson E, Li W, Sudre C, et al. Niftynet: a deep-learning platform for medical imaging [J]. Computer Methods and Programs in Biomedicine, 2018, 158: 113-122.

[145] Pawlowski N, Ktena S I, Lee M C H, et al. DLTK: state of the art reference implementations for deep learning on medical images [J]. CoRR, 2017, abs/1711.06853.

[146] MICHAEL BOONE. Nvidia introduces open-source project to accelerate computer vision cloud applications [EB/OL]. 2022. https://blogs.nvidia.com/blog/2022/09/20/computer-vision-cloud/.

[147] Valentin Kubarev, Victor Getmanskiy, Tim Allen. Portable, accelerated image processing with the oneapi image processing library [EB/OL]. 2022. https://www.intel.com/content/www/us/en/developer/articles/technical/portable-accelerated-image-processing-oneapi.html.

[148] Giduthuri R, Pulli K. Openvx: a framework for accelerating computer vision [C]//Mitra N J. SIGGRAPH ASIA 2016, Macao, December 5-8, 2016 - Courses. ACM, 2016: 14:1-14:50.

[149] Hsu K, Tseng H. Accelerating applications using edge tensor processing units [C]//de Supinski B R, Hall M W, Gamblin T. SC '21: The International Conference for High Performance Computing, Networking, Storage and Analysis, St. Louis, Missouri, USA, November 14 - 19, 2021. ACM, 2021: 56:1-56:14.

[150] Hu Y C, Li Y, Tseng H W. Tcudb: Accelerating database with tensor processors [C]// SIGMOD '22: Proceedings of the 2022 International Conference on Management of Data. New York, NY, USA: Association for Computing Machinery, 2022: 1360–1374.

[151] Dakkak A, Li C, Xiong J, et al. Accelerating reduction and scan using tensor core units [C]// ICS '19: Proceedings of the ACM International Conference on Supercomputing. New York, NY, USA: Association for Computing Machinery, 2019: 46–57.

[152] Liu X, Liu Y, Yang H, et al. Toward accelerated stencil computation by adapting tensor core unit on gpu [C]//ICS '22: Proceedings of the 36th ACM International Conference on Supercomputing. New York, NY, USA: Association for Computing Machinery, 2022.

[153] Groth S, Teich J, Hannig F. Efficient application of tensor core units for convolving images [C]//SCOPES '21: Proceedings of the 24th International Workshop on Software and Compilers for Embedded Systems. New York, NY, USA: Association for Computing Machinery, 2021: 1–6.

[154] Gartner, Inc. Gartner highlights 10 uses for ai-powered smartphones. [EB/OL]. 2018. https://www.gartner.com/en/newsroom/press-releases/2018-03-20-gartner-highlights-10-uses-for-ai-powered-smartphones.

[155] Stoica I, Song D, Popa R A, et al. A berkeley view of systems challenges for ai [J]. arXiv preprint arXiv:1712.05855, 2017.

[156] Qualcomm Technologies, Inc. Snapdragon profiler [EB/OL]. 2018. https://developer.qualcomm.com/software/snapdragon-profiler.

[157] Simpleperf [EB/OL]. 2018. https://developer.android.com/ndk/guides/simpleperf.

[158] Testmynet: Internet speed test. [EB/OL]. 2018. https://testmy.net/.

[159] Szegedy C, Vanhoucke V, Ioffe S, et al. Rethinking the inception architecture for computer vision [C]//2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016. IEEE Computer Society, 2016: 2818-2826.

[160] Ma N, Zhang X, Zheng H, et al. Shufflenet V2: practical guidelines for efficient CNN architecture design [C]//Ferrari V, Hebert M, Sminchisescu C, et al. Lecture Notes in Computer Science: volume 11218 Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIV. Springer, 2018: 122-138.

[161] Chen L, Zhu Y, Papandreou G, et al. Encoder-decoder with atrous separable convolution for semantic image segmentation [C]//Ferrari V, Hebert M, Sminchisescu C, et al. Lecture Notes in Computer Science: volume 11211 Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part VII. Springer, 2018: 833-851.

[162] Deng J, Dong W, Socher R, et al. Imagenet: A large-scale hierarchical image database [C]//2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA. IEEE Computer Society, 2009: 248-255.

[163] Everingham M, Gool L V, Williams C K I, et al. The pascal visual object classes (VOC) challenge [J]. Int. J. Comput. Vis., 2010, 88(2): 303-338.

[164] Zaremba W, Sutskever I, Vinyals O. Recurrent Neural Network Regularization [J]. ArXiv e-prints, 2014.

[165] Marcus M P, Marcinkiewicz M A, Santorini B. Building a large annotated corpus of english: The penn treebank [J]. Comput. Linguist., 1993, 19(2): 313-330.

[166] Hannun A Y, Case C, Casper J, et al. Deep speech: Scaling up end-to-end speech recognition [J]. CoRR, 2014, abs/1412.5567.

[167] Shi S, Wang Q, Xu P, et al. Benchmarking state-of-the-art deep learning software tools [C]// Cloud Computing and Big Data (CCBD), 2016 7th International Conference on. IEEE, 2016: 99-104.

[168] Snapdragon 820 mobile platform [EB/OL]. 2016. https://www.qualcomm.com/products/snapdragon-820-mobile-platform.

[169] Qualcomm Technologies, Inc. Oneplus 5t [EB/OL]. 2018. https://wiki.lineageos.org/devices/dumpling.

[170] Qualcomm Technologies, Inc. Oneplus 3 [EB/OL]. 2018. https://wiki.lineageos.org/devices/oneplus3.

[171] Redmi note 4x [EB/OL]. 2018. https://www.gsmarena.com/xiaomi_redmi_note_4x-8580.php.

[172] NVIDIA Corporation. Jetson tx2 module [EB/OL]. 2019. https://developer.nvidia.com/embedded/jetson-tx2.

[173] Guennebaud G, Jacob B, et al. Eigen v3 [EB/OL]. 2010. http://eigen.tuxfamily.org.

[174] Opencl-z android official webpage. [EB/OL]. 2015. http://web.guohuiwang.com/software/opencl_z_android.

[175] Snapdragon 835 benchmarks revealed: All you need to know about the new chip. [EB/OL]. 2017. https://www.trustedreviews.com/news/snapdragon-835-phones-processor-specs-speed-benchmark-chipset-cores-2944086.

[176] Qualcomm Technologies I. Qualcomm® snapdragontm mobile platform opencl general programming and optimization [M]. 2017.

[177] Huawei honor 10 [EB/OL]. 2018. https://www.amazon.com/Huawei-10-128GB-Factory-Unlocked-Smartphone/dp/B07D7GZBDW.

[178] Hiai foundation [EB/OL]. 2018. https://developer.huawei.com/consumer/cn/hiai#Foundation.

[179] Ignatov A, Timofte R, Kulik A, et al. Ai benchmark: All about deep learning on smartphones in 2019 [C]//2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW). 2019: 3617-3635.

[180] Qualcomm Corporation. Snapdragon 710 mobile platform [EB/OL]. 2022. https://www.qualcomm.com/products/application/smartphones/snapdragon-7-series-mobile-platforms/snapdragon-710-mobile-platform.

[181] Kedad-Sidhoum S, Monna F, Trystram D. Scheduling tasks with precedence constraints on hybrid multi-core machines [C]//2015 IEEE International Parallel and Distributed Processing

Symposium Workshop, IPDPS 2015, Hyderabad, India, May 25-29, 2015. IEEE Computer Society, 2015: 27-33.

[182] Topcuoglu H, Hariri S, Wu M y. Performance-effective and low-complexity task scheduling for heterogeneous computing [J]. IEEE transactions on parallel and distributed systems, 2002, 13(3): 260-274.

[183] Arabnejad H, Barbosa J G. List scheduling algorithm for heterogeneous systems by an optimistic cost table [J]. IEEE Transactions on Parallel and Distributed Systems, 2014, 25(3): 682-694.

[184] Witt C, Wheating S, Leser U. Los: Level order sampling for task graph scheduling on heterogeneous resources [C]//2018 IEEE/ACM Workflows in Support of Large-Scale Science (WORKS). 2018: 20-30.

[185] Kanemitsu H, Hanada M, Nakazato H. Clustering-based task scheduling in a large number of heterogeneous processors [J]. IEEE Transactions on Parallel and Distributed Systems, 2016, 27(11): 3144-3157.

[186] Augonnet C, Thibault S, Namyst R, et al. Starpu: a unified platform for task scheduling on heterogeneous multicore architectures [J]. Concurr. Comput. Pract. Exp., 2011, 23(2): 187-198.

[187] Glpk (gnu linear programming kit) [EB/OL]. 2020[2020-08-07]. https://www.gnu.org/software/glpk/.

[188] Szegedy C, Ioffe S, Vanhoucke V, et al. Inception-v4, inception-resnet and the impact of residual connections on learning [C]//Singh S P, Markovitch S. Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence. 2017: 4278-4284.

[189] Beaumont O, Canon L, Eyraud-Dubois L, et al. Scheduling on two types of resources: A survey [J]. ACM Comput. Surv., 2020, 53(3): 56:1-56:36.

[190] Hazelwood K, Bird S, Brooks D, et al. Applied machine learning at facebook: A datacenter infrastructure perspective [C]//2018 IEEE International Symposium on High Performance Computer Architecture (HPCA). 2018: 620-629.

[191] Grosser T, Groesslinger A, Lengauer C. Polly - performing polyhedral optimizations on a low-level intermediate representation [J]. Parallel Processing Letters, 2012, 22(04).

[192] Zhao J, Li B, Nie W, et al. Akg: Automatic kernel generation for neural processing units using polyhedral transformations [C]//PLDI 2021: Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation. New York, NY, USA: Association for Computing Machinery, 2021: 1233-1248.

[193] Harris M. Using shared memory in cuda c/c++ [EB/OL]. 2013. https://developer.nvidia.com/blog/using-shared-memory-cuda-cc/.

[194] Tianqi Chen. Working with operators using tensor expression [EB/OL]. 2022. https://tvm.apache.org/docs/tutorial/tensor_expr_get_started.html.

[195] Microsoft. Antares [EB/OL]. 2022. https://github.com/microsoft/antares/tree/latest.

[196] Sutskever I, Vinyals O, Le Q V. Sequence to sequence learning with neural networks [C]// NIPS'14: Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2. Cambridge, MA, USA: MIT Press, 2014: 3104-3112.

[197] Xie S, Girshick R, Dollár P, et al. Aggregated residual transformations for deep neural networks [C]//2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2017: 5987-5995.

[198] Liu Z, Lin Y, Cao Y, et al. Swin transformer: Hierarchical vision transformer using shifted windows [C]//2021 IEEE/CVF International Conference on Computer Vision, ICCV 2021, Montreal, QC, Canada, October 10-17, 2021. IEEE, 2021: 9992-10002.

[199] Bondhugula U, Acharya A, Cohen A. The pluto+ algorithm: A practical approach for parallelization and locality optimization of affine loop nests [J]. ACM Trans. Program. Lang. Syst., 2016, 38(3): 12:1-12:32.

[200] Multi-Level IR Compiler Framework committee. 'affine' dialect [EB/OL]. 2022. https://mlir.llvm.org/docs/Dialects/Affine/.

[201] Mark Harris and Kyrylo Perelygin. Cooperative groups: Flexible cuda thread programming [EB/OL]. 2017. https://developer.nvidia.com/blog/cooperative-groups/.

[202] Rocha R C, Petoumenos P, Wang Z, et al. Effective function merging in the ssa form [C]// Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation. 2020: 854-868.

[203] Ma J, Zhao Z, Yi X, et al. Modeling task relationships in multi-task learning with multi-gate mixture-of-experts [C]//KDD '18: Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining. New York, NY, USA: Association for Computing Machinery, 2018: 1930–1939.

[204] Pham H, Guan M Y, Zoph B, et al. Efficient neural architecture search via parameter sharing [C]//Dy J G, Krause A. Proceedings of Machine Learning Research: volume 80 Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018. PMLR, 2018: 4092-4101.

[205] NVIDIA Corporation. Nvidia nsight compute [EB/OL]. 2022. https://developer.nvidia.com/nsight-compute.

[206] Cheng Y, Wang D, Zhou P, et al. A survey of model compression and acceleration for deep neural networks [J]. CoRR, 2017, abs/1710.09282.

[207] Rajbhandari S, Ruwase O, Rasley J, et al. Zero-infinity: Breaking the gpu memory wall for extreme scale deep learning [C]//Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. 2021: 1-14.

[208] Wang Z, O' Boyle M. Machine learning in compiler optimization [J]. Proceedings of the IEEE, 2018, 106(11): 1879-1901.

[209] Turner J, Crowley E J, O'Boyle M F. Neural architecture search as program transformation exploration [C]//Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems. 2021: 915-927.

[210] Liba O, Movshovitz-Attias Y, Cai L, et al. Sky optimization: Semantically aware image processing of skies in low-light photography [C]//2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR Workshops 2020, Seattle, WA, USA, June 14-19, 2020. Computer Vision Foundation / IEEE, 2020: 2230-2238.

[211] Hasinoff S W, Sharlet D, Geiss R, et al. Burst photography for high dynamic range and low-light imaging on mobile cameras [J]. ACM Trans. Graph., 2016, 35(6): 192:1-192:12.

[212] Liu Y, Lai W, Chen Y, et al. Single-image HDR reconstruction by learning to reverse the camera pipeline [C]//2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2020, Seattle, WA, USA, June 13-19, 2020. Computer Vision Foundation / IEEE, 2020: 1648-1657.

[213] Heidarizadeh A. Preprocessing methods of lane detection and tracking for autonomous driving [J]. CoRR, 2021, abs/2104.04755.

[214] Liu P, Zhang C, Qi H, et al. Multi-attention densenet: A scattering medium imaging optimization framework for visual data pre-processing of autonomous driving systems [J]. IEEE Transactions on Intelligent Transportation Systems, 2022: 1-12.

[215] de Aguiar Neto F S, da Costa A F, Manzato M G, et al. Pre-processing approaches for collaborative filtering based on hierarchical clustering [J]. Information Sciences, 2020, 534: 172-191.

[216] Johnson J, Douze M, Jégou H. Billion-scale similarity search with gpus [J]. IEEE Trans. Big Data, 2021, 7(3): 535-547.

[217] Zhou X, Du Z, Guo Q, et al. Cambricon-s: Addressing irregularity in sparse neural networks through a cooperative software/hardware approach [C]//2018 51st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO). 2018: 15-28.

[218] Zhao Y, Du Z, Guo Q, et al. Cambricon-f: Machine learning computers with fractal von neumann architecture [C]//ISCA '19: Proceedings of the 46th International Symposium on Computer Architecture. New York, NY, USA: Association for Computing Machinery, 2019: 788-801.

[219] Samuel Fletcher. 300+ internet usage statistics for 2022 [EB/OL]. 2022. https://supplygem.com/internet-usage-statistics/.

[220] Google Inc. Edge tpu compiler [EB/OL]. 2022. https://coral.ai/docs/edgetpu/compiler/.

[221] Lee J, Chirkov N, Ignasheva E, et al. On-device neural net inference with mobile gpus [J]. CoRR, 2019, abs/1907.01989.

[222] Huawei Inc. Cann v100r020c20 application software development guide [EB/OL]. 2021. https://support.huawei.com/enterprise/en/doc/EDOC1100180746/b6db6b40/ascendcl-overview.

[223] Wu J, Leng C, Wang Y, et al. Quantized convolutional neural networks for mobile devices [C]//2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR). 2016: 4820-4828.

[224] Shin S, Hwang K, Sung W. Fixed-point performance analysis of recurrent neural networks [C]//2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP). 2016: 976-980.

[225] Hubara I, Courbariaux M, Soudry D, et al. Quantized neural networks: Training neural networks with low precision weights and activations [J]. J. Mach. Learn. Res., 2017, 18(1): 6869–6898.

[226] Canny J. A computational approach to edge detection [J]. IEEE Trans. Pattern Anal. Mach. Intell., 1986, 8(6): 679–698.

[227] Beauchemin S S, Barron J L. The computation of optical flow [J]. ACM Comput. Surv., 1995, 27(3): 433–466.

[228] Jia Z, Padon O, Thomas J, et al. Taso: Optimizing deep learning computation with automatic generation of graph substitutions [C]//SOSP '19: Proceedings of the 27th ACM Symposium on Operating Systems Principles. New York, NY, USA: Association for Computing Machinery, 2019: 47-62.

[229] Narkowicz K. ACES Application Color Management Reference [EB/OL]. 2016. https://knarkowicz.wordpress.com/2016/01/06/aces-filmic-tone-mapping-curve/.

[230] Arrighetti W. The academy color encoding system (aces): A professional color-management framework for production, post-production and archival of still and motion pictures [J]. Journal of Imaging, 2017, 3(4).

[231] Liu L, Gu J, Lin K Z, et al. Neural sparse voxel fields [C]//Larochelle H, Ranzato M, Hadsell R, et al. Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual. 2020.

[232] Müller T, Evans A, Schied C, et al. Instant neural graphics primitives with a multiresolution hash encoding [J]. ACM Trans. Graph., 2022, 41(4): 102:1-102:15.

[233] Amos B, Ludwiczuk B, Satyanarayanan M. Openface: A general-purpose face recognition library with mobile applications [R]. CMU-CS-16-118, CMU School of Computer Science, 2016.

[234] Bradbury J, Frostig R, Hawkins P, et al. JAX: composable transformations of Python+NumPy programs [CP/OL]. 2018. http://github.com/google/jax.

致 谢

在计算所攻读博士学位的近七年岁月，如白驹过隙，忽然而已。回顾这七年的时光，很幸运能够体验了一段如此丰富多彩的科研生活。

首先我要感谢我的导师崔慧敏研究员。崔老师是我们心目中的学术女神。在我熬夜投论文时帮我耐心地修改文章，迷茫时为我答疑解惑，失落时鼓励我继续前行。您对科研方向的高瞻远瞩与对科研工作热忱的投入激励我刻苦钻研。您总是能够提出新的思路，鼓励我打破束缚勇敢地尝试。您给了我极大的支持与自由，让我无拘无束的做感兴趣的研究方向；您还给予我很多机会参加国内外学术会议和项目交流，极大地拓展了我的视野与阅历；在生活中，您也给与了我许多无私的帮助和呵护。本文的顺利完成离不开崔老师的悉心指导与帮助。感谢您一直以来对我的帮助与包容！谢谢您！

感谢冯晓兵研究员。冯老师总是能够给我提出许多宝贵的指导与建议，指引我科研前进的方向。您总是教导我们不能唯论文，更重要的是做的事情的意义和价值。在您身上我感受到了老一辈科学家科研为国为民的情怀，鼓励我脚踏实地做实事。除此之外，冯老师严谨治学的态度，对后辈无微不至的关怀也深深感染了我。在此，向冯老师表达我最衷心的感谢！

感谢赵家程副研究员。赵老师是我的朋友也是老师，更是我一直学习的榜样。赵老师是我的领路人，帮我调试了许多的 bug，陪我熬过一个又一个的通宵，讨论了数不清的科研问题。赵老师极强的动手能力、对问题探究到底的态度极大的影响了我。在生活上赵老师带我打球玩耍，困难时给了我许多的帮助。

感谢清华大学的翟季冬老师、信工所的霍玮老师、新南威尔士大学的薛京灵老师、利兹大学的王峥老师、阿伯丁大学的温源老师和清醒异构的余腾老师对我研究课题提供的指导与帮助。

感谢实验室的小伙伴们对我的帮助与陪伴，让我从小师弟成长为实验室的大师兄，他们是：王晨曦、阮功、宋育庚、石洋、张亚琳、李登辉、李奕瑾、王智慧、张馨元、刘子娟、耿洪娜、王召德、高猛、杨永新、陈磊、肖佳伟、张阳煜、李帅江、汪诗洋、李权熹、李梓豪、李志成。与你们一起让枯燥乏味的读博岁月充满欢声笑语。感谢一起打球的兄弟们，感谢博士期间室友的支持。

特别感谢我的家人给我无私的支持。感谢我的父亲支撑起我们的家庭，感谢我的母亲默默无闻的付出。你们对我毫无保留的爱支撑我度过了最艰难的岁月，让我从贫困的农村走向了广阔的世界。

感谢我的女朋友陈潇潇，你让我知道了什么是双向奔赴的爱情。读博期间有你的陪伴是我最大的幸运，余生打怪升级的路上还请多多指教。

2022 年 12 月

作者简历及攻读学位期间发表的学术论文与其他相关学术成果

作者简历：

夏春伟，河南省鹤壁市人。

2012 年 09 月——2016 年 07 月，在天津大学计算机科学与技术学院获得学士学位。

2016 年 09 月——2022 年 12 月，在中国科学院计算技术研究所攻读博士学位。

已发表（或正式接受）的学术论文：

(1) **Chunwei Xia**, Jiacheng Zhao, Huimin Cui , Xiaobing Feng, HOPE: A Heterogeneity-Oriented Parallel Execution Engine for Inference on Mobiles, High Technology Letters, 2022

(2) **Chunwei Xia**, Jiacheng Zhao, Huimin Cui , Xiaobing Feng and Jingling Xue, DNNTune: Automatic Benchmarking DNN Models for Mobile-cloud Computing, in ACM Transactions on Architecture and Code Optimization（ACM TACO）2019

(3) **Chunwei Xia**, Jiacheng Zhao, Huimin Cui , Xiaobing Feng, Characterizing DNN Models for Edge-Cloud Computing, in IEEE International Symposium on Workload Characterization (IEEE IISWC) 2018 (Poster)

(4) Ning Lin, Xiaoming Chen, **Chunwei Xia**, Jing Ye, Xiaowei Li，ChaoPIM: A PIM-based Protection Framework for DNN Accelerators Using Chaotic Encryption，Asian Test Symposium，（ATS）,2021

(5) Jiacheng Zhao, Yisong Chang, Denghui Li, **Chunwei Xia**, Huimin Cui, Ke Zhang, Xiaobing Feng, On Retargeting the AI Programming Framework to New Hardwares, IFIP International Conference on Network and Parallel (NPC) 2018

已投稿的学术论文：

(1) **Chunwei Xia**, Jiacheng Zhao, Zheng Wang, Yuan Wen, Teng Yu, Huimin Cui, Xiaobing Feng, Think Big, Act Small: Optimizing Deep Learning Inference via Global Analysis and Local Transformation, submitted to IEEE/ACM International Symposium on Code Generation and Optimization (CGO 2023)

申请或已获得的专利：

夏春伟，赵家程，崔慧敏，冯晓兵，在异构处理单元上执行深度神经网络的方法，申请号：202010493830.8

参加的研究项目：

(1) 华为-计算所合作项目“方舟编译器生态关键技术研究”，2020 年 1 月 ~ 2021 年 6 月

(2) 国家重点研发计划“面向异构融合数据流加速器的编程模型及编译器优化”，2016 年 6 月 ~ 2020 年 12 月

获奖情况：

(1) 中国科学院大学一等博士学业奖学金，2920 年

(2) 中国科学院大学“三好学生”2019 年